User Manual

AWTK Designer MVVM 使用指南

用户手册

V1.0.1 Date:2025/10/31

类别	内容
关键词	AWTK Designer、MVVM-C、MVVM-JS、案例分析
摘要	本文介绍了如何使用 AWTK Designer 制作 MVVM 项目，并包含 MVVM-C 项目和 MVVM-JS 项目的案例分析。

修订历史

版本	日期	原因
1.0.1	2023/01/12	• add Advance_Usages
1.0.0	2022/06/16	• first implement

目 录

文档导读

本文介绍了如何使用 AWTK Designer 制作 MVVM 项目，并包含 MVVM-C 项目和 MVVM-JS 项目的案例分析，可以帮助读者了解 AWTK Designer 中 MVVM 的相关功能并快速上手，制作属于自己的 MVVM 项目。

1. 简介

1.1 AWTK

AWTK 全称 Toolkit AnyWhere，是 ZLG 基于 C 语言开发的开源 GUI 引擎，详情可前往 AWTK 官网[1] 查看。源码可前往 GitHub 仓库[2] 下载。

在阅读本文之前，首先需要对 AWTK 有基本用法有所了解，具体可以参阅《AWTK 开发实践》。

1.2 MVVM 模式

在开发应用程序时，要把用户界面和业务逻辑分离开来，这是每个程序员都知道的常识。分离用户界面和业务逻辑有几个重要的好处：

1. 有利于隔离变化。当界面发生变化时，减少业务逻辑代码的改动，降低开发的成本和周期。
2. 有利于自动测试。分离用户界面和业务逻辑是提高代码可测试性的重要手段，让编写单元测试程序成为可能。
3. 有利于分工合作。界面和业务逻辑的分离有助于设计师和程序员分工协作，减少矛盾，提高开发效率。

目前最常用的分离用户界面和业务逻辑模式就是 MVVM 模式，它把系统分成模型（Model）、视图（View）和视图模型（ViewModel）这三部分，明确的把用户界面和业务逻辑分离开来，如下图所示：

图 1.1 MVVM 模式

1.2.1 模型（Model）

模型（Model）就是业务逻辑。我们经常说面向对象的设计和建模，所建立的模型就是这里的模型，是对现实世界中业务逻辑的抽象。

面向对象的建模，就是要找到业务逻辑中有哪些类以及这些类之间的关系。类描述了对象的属性和行为，类是设计时的概念，对象是运行时的概念。对象通常就是业务实体，Model 包含一个或多个业务实体。

需要注意的是，Model 并不单纯指的是数据，它是对象的集合，对象是即有数据又有行为的，光有数据没有行为，就无法让对象之间协作，无法共同完成业务逻辑。

[1] https://www.zlg.cn/index/pub/awtk.html
[2] https://github.com/zlgopen/awtk

1.2.2 视图（View）

视图（View）就是用户与软件交互的界面，对于 GUI 应用程序来讲，就是窗口和对话框，以及上面的各种控件。视图应该说是为用户提供了一个界面，让用户可以观察和操作模型。

1.2.3 视图模型（ViewModel）

在 MVVM 模式中，View 与 Model 之间通过一些规则建立联系，也就是**数据绑定**和**命令绑定**：

- View 通过数据绑定在界面上显示 Model 中的指定数据，当 Model 中的数据更新时又会通知 View 进行更新。
- View 通过命令绑定来执行 Model 中的指定操作，即用户在界面上触发事件时（比如点击按钮），执行 Model 业务实体对象中的函数（行为）。

根据**数据绑定**和**命令绑定**规则实现 View 和 Model 交互的中间层就是视图模型（View-Model），它作为二者沟通的桥梁，当用户在 View 上修改数据时将其自动同步到 Model 中，当 Model 中的数据有变化时自动更新到 View 上，并且通过命令绑定实现 View 中触发事件执行 Model 层函数的功能。

注: MVVM 模式实现的核心就是一条条数据绑定和命令绑定的规则，对规则的处理和解释成了 MVVM 框架或公用库，可以在多个项目中共享，即下文所说的 AWTK-MVVM[3]。

1. ViewModel 与 View 的关系

由于 ViewModel 在 View 眼中仍然是模型，它与 View 没有直接关系，因此是可以为 ViewModel 编写单元测试的。

2. ViewModel 与 Model 的关系

虽然在 View 眼中 ViewModel 仍然是模型，但它其实与 Model 并不相同，之所以要引入 ViewModel，主要原因如下：

- Model 中的数据有时并不适合直接显示在 View 上。比如出版日期在 Model 中可能是是整数，直接显示出来，用户就看不懂，需要转换成人类可以理解的字符串。
- Model 中的一个数据项可能以多种形式呈现在 View 上。比如出版日期除了直接显示外，也可能给最近出版的书显示一个新书标志。
- View 中的输入数据和 Model 中的存储数据，它们的格式有时可能不同。比如出版日期，View 上输入的格式和 Model 里存储的格式就不一样。
- View 需要 ViewModel 提供数据校验规则来判断视图上的输入是否合法。
- 有些数据是有依赖关系的，这些处理可以交给 ViewModel 去做。比如在输入收货地址时，选择省份时，城市列表跟着变化。
- 有些状态不属于业务逻辑，但是需要保存下来。如为了支持撤销操作，需要保存命令历史记录。
- 在 C/C++ 等静态语言中，没有办法通过函数的名称去调用对象的成员函数，也没有办

[3] https://github.com/zlgopen/awtk-mvvm

法通过属性的名称去访问对象的成员变量。ViewModel 需要提供一种机制来实现这些功能，否则绑定规则就没法实现。

可以看出 ViewModel 既是 View 和 Model 之间的粘合剂，又是润滑剂。

1.2.4 MVVM 模式的优缺点

MVVM 模式有以下优点：

1. 强制分离用户界面和业务逻辑。MVVM 中不需要编写界面相关的代码，自然没有办法让用户界面和业务逻辑耦合到一起。
2. 界面描述文件和程序代码之间具有更松的耦合。MVVM 模式采用了面向意图的编程，界面上表现的只是一种意图，至于在程序中，谁去实现，怎么实现，甚至有没有实现，界面都是不关心的，所以界面和代码之间值的耦合是很松的。
3. 不需要学习 GUI 的 API。通过数据绑定和命令绑定建立视图和模型之间的联系。用声明式的规则取代命令式的代码，不需要写界面相关的代码，所以不需要学习 GUI 的 API。

除了以上优点，MVVM 模式还存在以下缺点，在开发应用前需要根据项目实际情况判断是否适合采用 MVVM 模式：

1. 内存问题。据说 WPF(Windows Persentation Foundation) 内存开销很大，即使在 PC 上也大得让人无法忍受 (常有人借此来反对 MVVM)。WEB 前端流行的 MVVM 框架，像 React 和 Vue.js 等，采用虚拟 DOM 的方式，内存需求可能有大幅度增加。
2. 性能问题。MVVM 通常的做法是，模型有变化时会刷新整个界面。如果采用虚拟 DOM 的方式，还需要重新构建一个虚拟 DOM，与原来的虚拟 DOM 比较，性能开销就更大了。
3. 调试问题。数据绑定使得 Bug 很难被调试，它会将 Bug 转移到其他位置，比如看到界面显示异常，有可能是 View 有问题，也可能是 Model 有问题，或者二者都有问题。

注: 更多关于 MVVM 模式的相关介绍详见：awtk-mvvm/docs/8.intro.md[4]。

1.3 AWTK-MVVM

AWTK-MVVM[5] 是一套用 C 语言开发的，专门为嵌入式平台优化的 MVVM 框架。它实现了数据绑定、命令绑定和窗口导航等基本功能，使用 AWTK-MVVM 开发应用程序，无需学习 AWTK 本身的 API，只需学习绑定规则和模型的实现方式即可。

与其它 MVVM 框架相比，AWTK-MVVM 特点有：

- 代码小。
  - 核心代码：约 3000 行。
  - AWTK 相关代码：约 700 行。
  - JerryScript 相关代码 (可选)：约 1700 行。
- 性能高。核心代码用 C 语言开发，其性能与直接使用 AWTK 差别不大。
- 内存开销小。

[4] https://github.com/zlgopen/awtk-mvvm/blob/master/docs/8.intro.md
[5] https://github.com/zlgopen/awtk-mvvm

  - 一条绑定规则约占 150 字节。一个窗口上若有 10 条绑定规则，内存占用也就 1.5K。
  - 列表中的绑定规则可以共享，额外增加的内存开销，比例也是很小的。
- 隔离更彻底。在 AWTK-MVVM 中，界面和绑定规则在 XML 中描述，开发应用程序时只需要开发模型 (也就是业务逻辑) 即可。除非通过特殊手段，开发者没有机会直接去访问视图。
- 易调试。
  - 提供 ViewModel 的框架生成工具，避免手写代码造成的错误。
  - 增加了各种异常处理的日志信息。
  - 提供大量的示例程序以供参考。
  - 开放源码，调试方便。
- 支持多语言开发。
  - 目前支持 C 语言和 JS 语言。
  - 以后会根据需要支持新的编程语言。

注: AWTK-MVVM[6] 的内存需求评估请参考：awtk-mvvm/docs/memory_req.md[7]

1.4 AWTK Designer 中的 MVVM 项目

AWTK Designer 是专门用来制作 AWTK 应用程序 UI 界面的实用工具。只要通过拖拽和点击就可以完成复杂的界面设计，操作简单；可以随时预览效果，所见即所得，下载地址和使用方法详见《AWTK Designer 用户手册》。

AWTK Designer 专业版中内置了 AWTK-MVVM[8]，可以新建并编辑 MVVM 项目，降低 MVVM 项目的开发难度，提高开发效率。目前支持以下两类 MVVM 项目，它们都使用 XML 描述 UI，如下图所示：

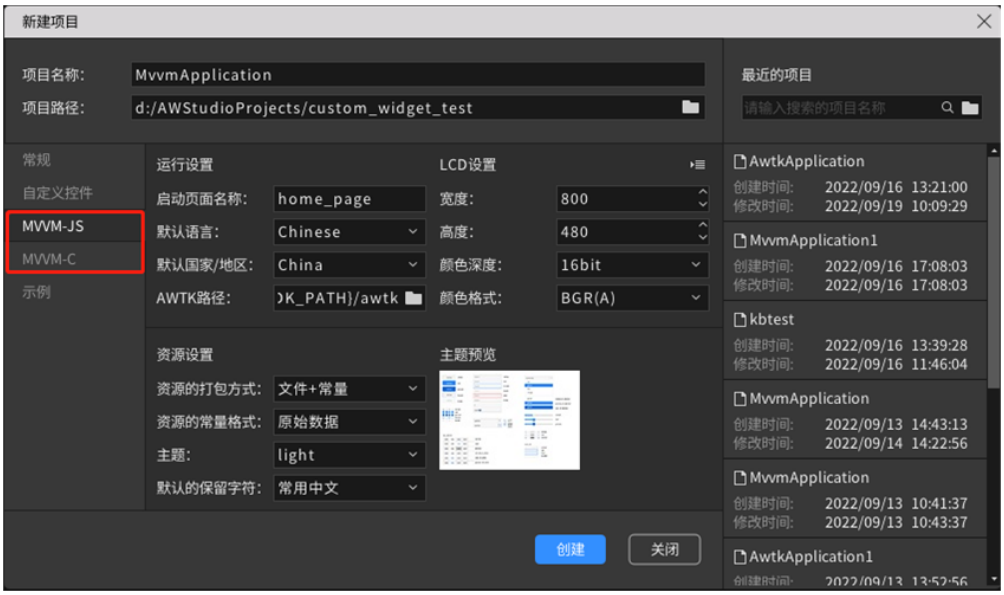

图 1.2 AWTK Designer 中的 MVVM 项目

[6] https://github.com/zlgopen/awtk-mvvm
[7] https://github.com/zlgopen/awtk-mvvm/blob/master/docs/memory_req.md
[8] https://github.com/zlgopen/awtk-mvvm

- MVVM-C 项目：使用 AWTK-MVVM 框架，基于 C 语言开发应用程序的工程项目，该类项目的 Model 层和 ViewModel 层均采用 C 语言实现。
- MVVM-JS 项目：使用 AWTK-MVVM 框架和 JerryScript 引擎，基于 JS 语言开发应用程序的工程项目，该类项目的 Model 层和 ViewModel 层均采用 JS 语言实现。

在后面的章节中，我们将详解介绍如何使用 AWTK Designer 制作 MVVM-C 项目和 MVVM-JS 项目。

2. 制作 MVVM-C 项目

MVVM-C 项目是使用 **AWTK-MVVM** 框架和 C 语言开发应用程序的一个工程项目，关于 MVVM 的介绍，请参阅 AWTK-MVVM[9] 中的文档。下面详细介绍如何使用 AWTK Designer 制作一个 MVVM-C 项目。

2.1 新建 MVVM-C 项目工程

AWTK Designer 启动后默认打开”新建项目”对话框，如下图所示。项目类型选择 MVVM-C，然后设置好项目的相关参数后，最后点击”创建”按钮即可创建一个 MVVM-C 项目，关于新建项目工程的更多信息，请参阅《AWTK Designer 用户手册》。

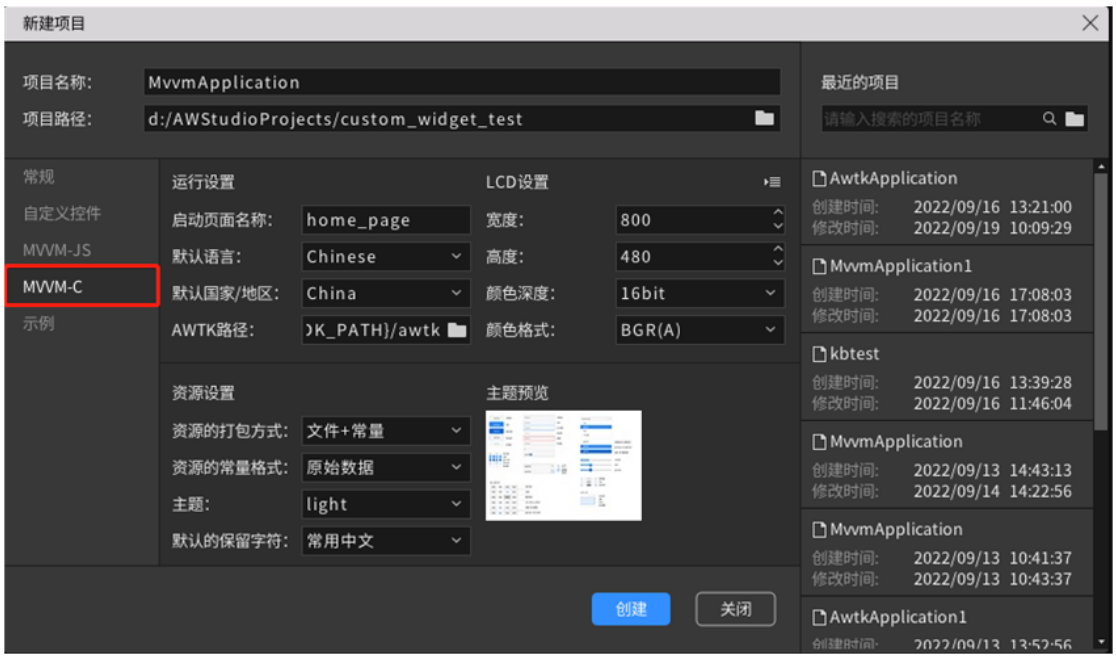

图 2.1 新建 MVVM-C 项目

注: 点击主界面工具栏的”新建项目”按钮，也可打开”新建项目”对话框。

新建项目后，AWTK Designer 会进入主界面并默认创建和打开一个名为 home_page 的页面，这个页面就可以称为一个 View（视图）。

对比常规项目，主界面增加了下图框选的内容，继续阅读下文便知其用途，关于 AWTK Designer 界面的更多信息，请参阅《AWTK Designer 用户手册》。

[9] https://github.com/zlgopen/awtk-mvvm

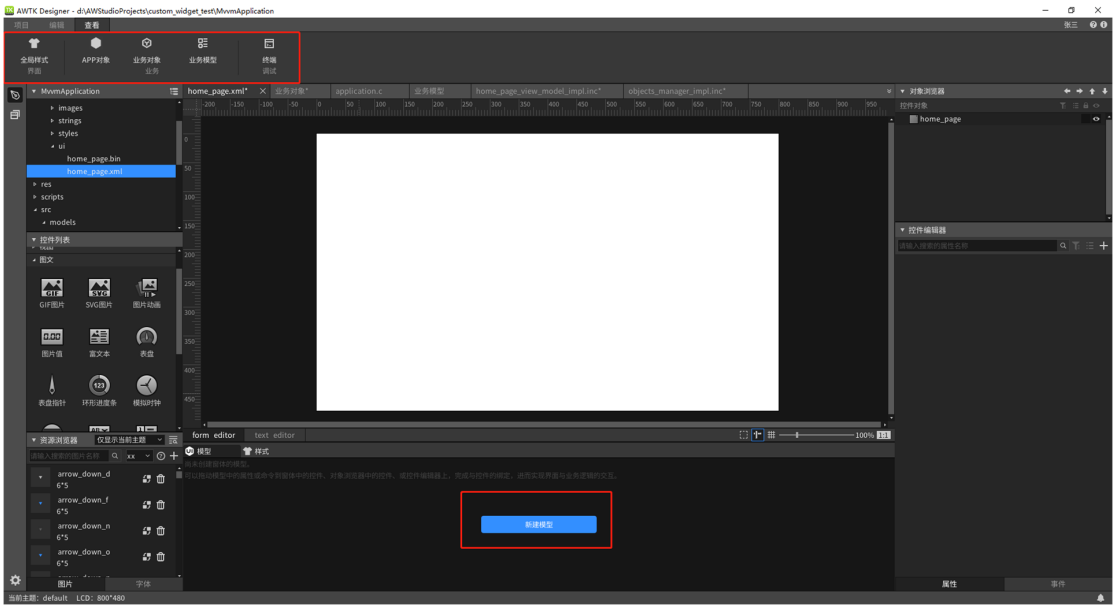

图 2.2　主界面

项目的目录结构

点击下图框选按钮切换到常规模式，可以看到项目的目录结构：

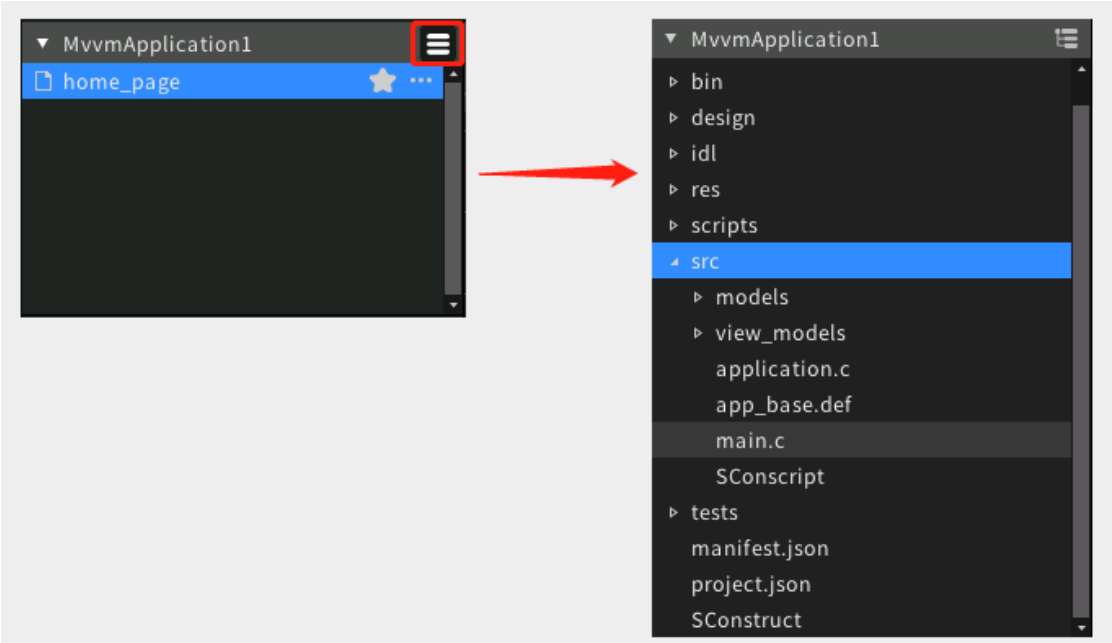

图 2.3　项目目录结构

对比常规项目，MVVM-C 项目会多出 models 和 view_models 两个文件夹，models 文件夹用于存放模型的 C 代码，models 文件夹中默认会有一个全局对象管理器 objects_manager，它是一个特殊的 Model（模型），提供访问业务数据和操作的接口，一个程序有且仅有一个，被 ViewModel（视图模型）共享，默认在程序启动时创建、退出时销毁。而 view_models 文件夹用于存放视图模型的 C 代码。

注：AWTK Designer 新建的常规项目目录结构的详细信息，请参阅《AWTK Designer 用户手册》。

2.2 ViewModel（视图模型）

ViewModel（视图模型，简称 VM）是与视图沟通的一座桥梁，想要完成数据绑定和命令绑定，就必须关联视图模型和视图。

数据绑定是通过规则在 View 和 Model 之间建立联系，当用户在 View 上修改数据时，View 上的数据自动同步到 Model 中，当 Model 中的数据有变化时，自动更新到 View 上去。而命令绑定是通过规则在 View 中控件的事件和 Model 之间建立联系，用户在 View 上触发某个事件之后，框架自动执行 Model 中对应的函数。

2.2.1 为 View 新增 ViewModel

现在为 home_page 页面增加一个 VM 。

步骤一：点击“模型”->“新建模型”按钮新增 VM：

图 2.4 点击新增模型按钮

步骤二：之后会弹出如下图的对话框，勾选“添加全局对象为 ViewModel 的属性”会把全局对象管理器对象添加到这个新增的 VM 中，点击确定就会为 home_page 新增一个 VM 。

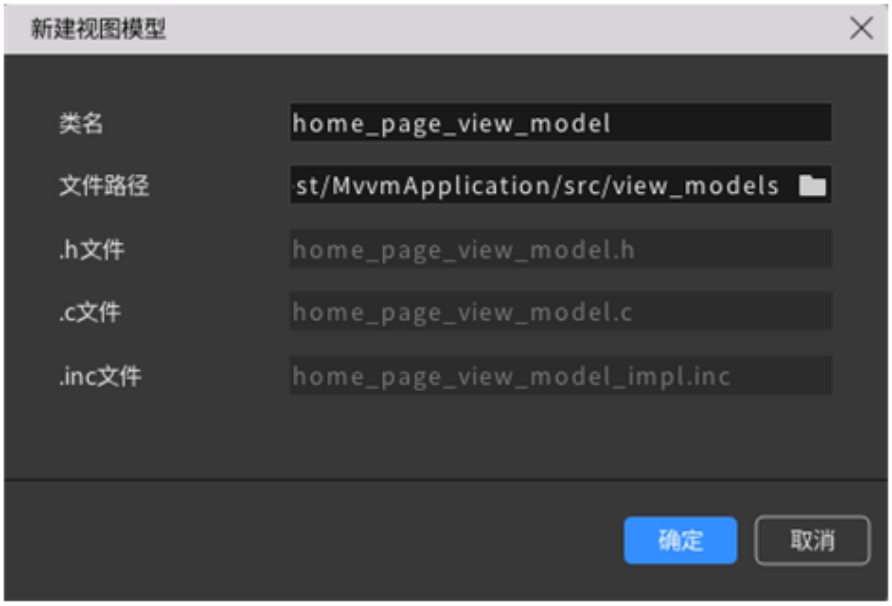

图 2.5 弹出对话框

同时 AWTK Designer 默认会在项目 src/view_models 目录下自动生成 VM 代码（切换至常规模式可以看到目录结构）：

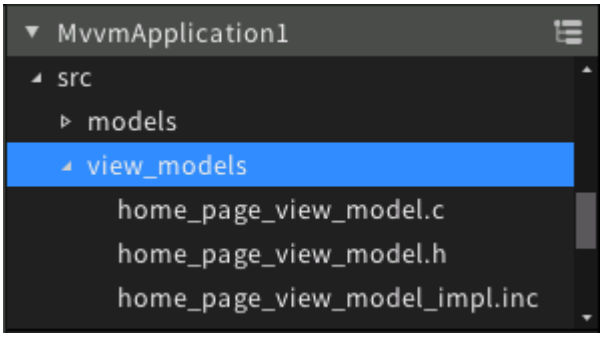

图 2.6　自动增加代码

并且在 src 目录下 application.c 自动增加注册 VM 的代码：

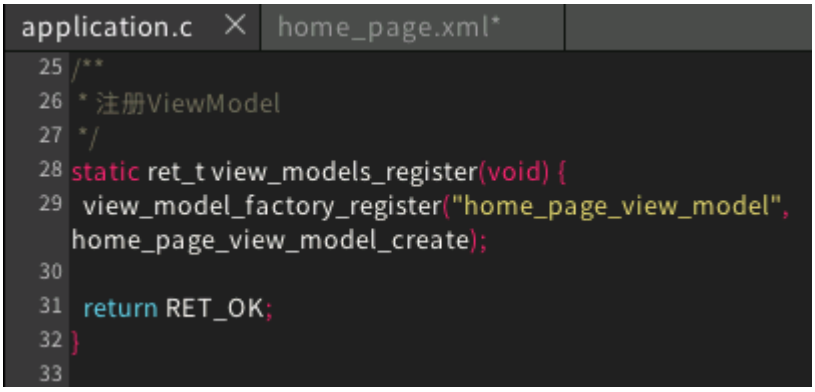

图 2.7　自动增加代码

2.2.2　为 ViewModel 添加属性

在数据绑定前，需要先为 VM 添加属性。属性分为局部属性和全局属性，在窗口关闭，与之关联的 VM 跟着销毁时，局部属性则会跟着一起销毁，而全局属性则不会销毁，直到程序结束。

1.　为 ViewModel 添加局部属性

现在为 home_page_view_model 添加一个局部属性。点击“模型”->“属性”->“+”按钮，之后就会在 VM 中增加一个局部属性：

图 2.8　添加局部属性

2. 编辑属性

增加属性后还需要编辑属性中名称、类型、默认值和描述，鼠标左键点击属性中的类型可对其进行选择：

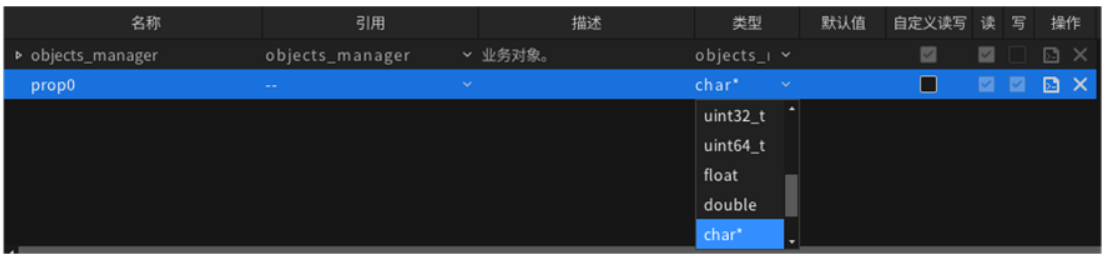

图 2.9 编辑属性

鼠标左键双击属性中的名称、默认值和描述可对其进行编辑：

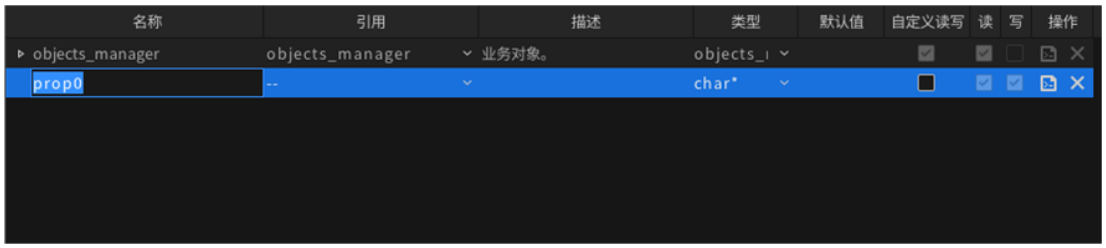

图 2.10 编辑属性

通过上述步骤，就可以得到一个已编辑的属性了：

图 2.11 编辑属性完成

3. 为 ViewModel 添加全局属性

现在为 home_page_view_model 添加一个全局属性（为 VM 添加属性后，对应的 VM 代码也会跟着添加属性）。

（1）添加属性到全局对象管理器

步骤一：在菜单栏点击“查看”->“业务对象”按钮：

图 2.12 点击全局对象按钮

步骤二：跳转到业务对象窗口，点击“属性”-> “+”按钮，之后就会在全局对象管理器 objects_manager 中增加一个属性（在 objects_manager 中增加一个属性，在项目 src/models 目录下 objects_manager 的代码也会跟着改变。）：

图 2.13　添加属性到全局对象管理器

步骤三：按照上文编辑得到一个已编辑的全局属性 g_prop：

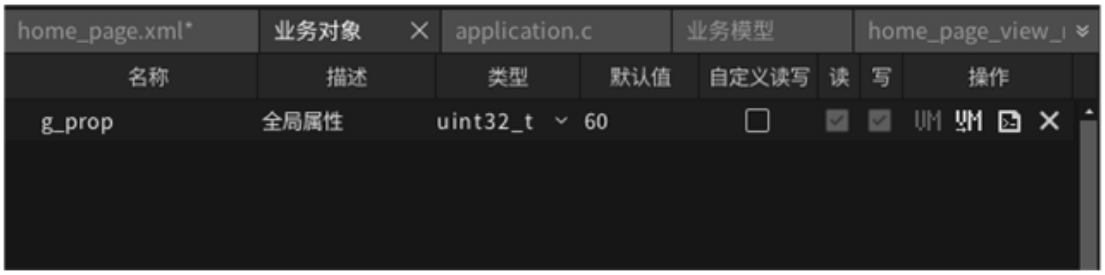

图 2.14　编辑属性完成

（2）添加全局属性到 ViewModel

回到 home_page 界面，点击全局对象管理器 object_manager 就会发现多出一个子属性 g_prop：

图 2.15　添加全局属性

在业务对象窗口中点击下图中框选按钮，之后弹出对话框，添加 object_manager.g_prop 到 VM 即可：

图 2.16　添加全局属性到视图模型

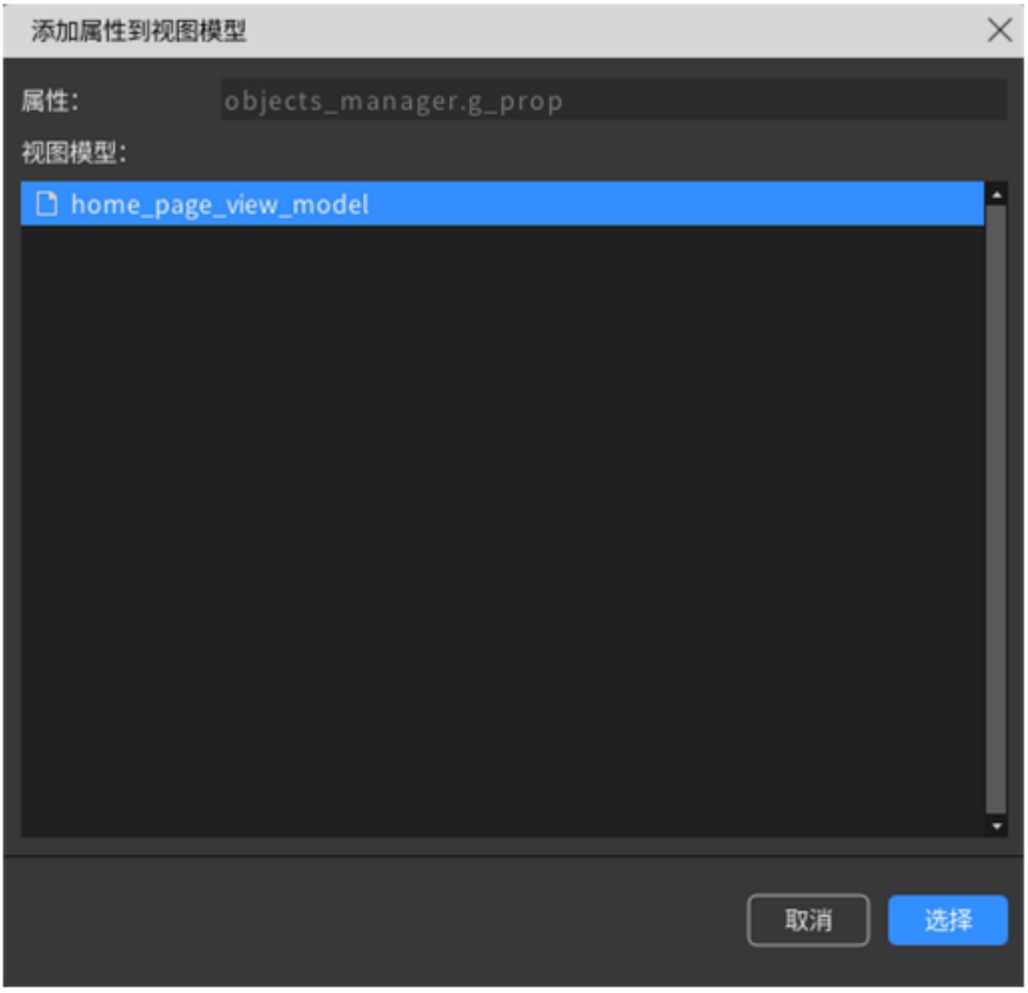

图 2.17　添加全局属性到视图模型

回到 home_page 界面，就会发现多出一个属性 g_prop：

图 2.18　添加全局属性到视图模型

2.2.3　数据绑定

下面会以上文中为 home_page_view_model 添加的 property 属性绑定到控件 slider 的 value 属性为例，讲解如何完成数据绑定。

1.　增加数据绑定规则

方式一：

步骤一：选择需要绑定数据的控件，然后选择需要绑定的控件属性，点击框选按钮：

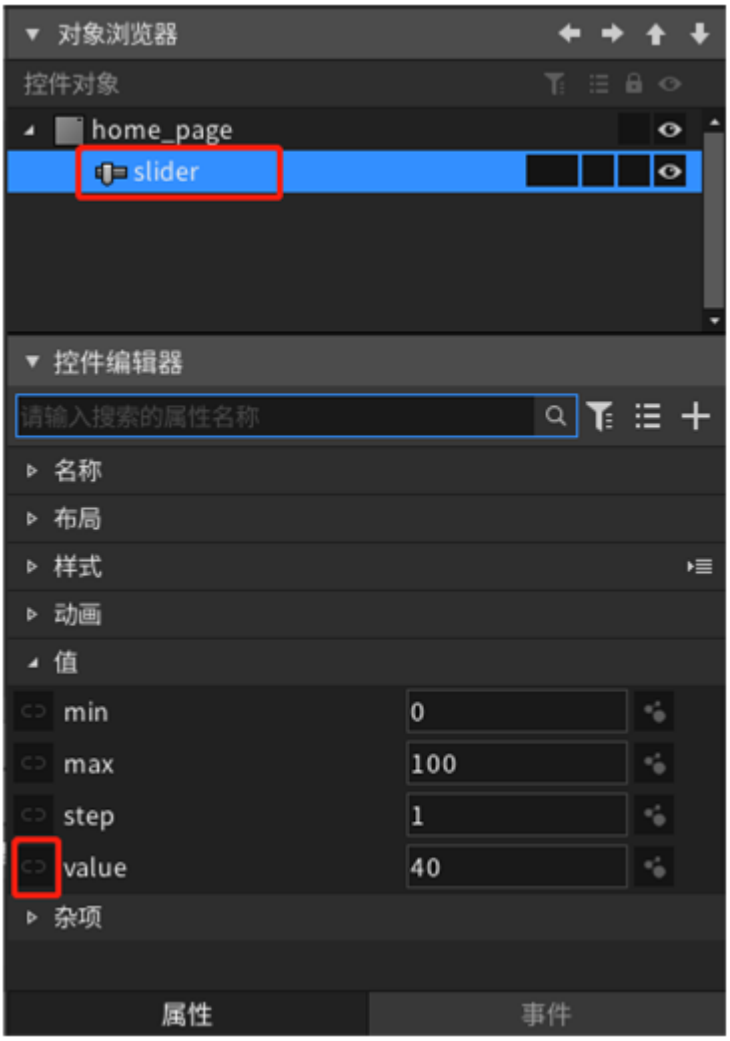

图 2.19　数据绑定步骤一

步骤二：弹出选项，选择设置属性绑定：

图 2.20　数据绑定步骤二

步骤三：弹出数据绑定编辑对话框，在”绑定的数据”一栏选择需要绑定的属性或者在编辑器中输入属性，也可以点击右侧的按钮进入设置绑定规则对话框，然后点击确定：

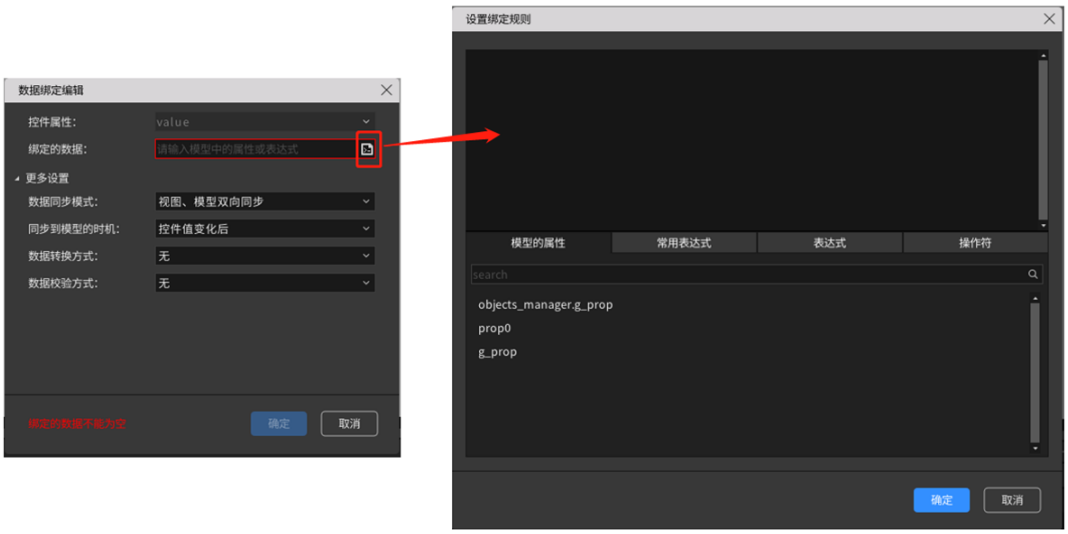

图 2.21　数据绑定步骤三

上文只是使用了最简单的绑定规则，还可以根据自身的需求去选择模式，使用内嵌表达式和操作符。点击按钮就会添加规则，在按钮上停留还会告诉其作用是什么，还有使用的方法。想了解更多的绑定规则请参阅 AWTK-MVVM 数据绑定[10]。

[10] https://github.com/zlgopen/awtk-mvvm/blob/master/docs/10.data_binding.md

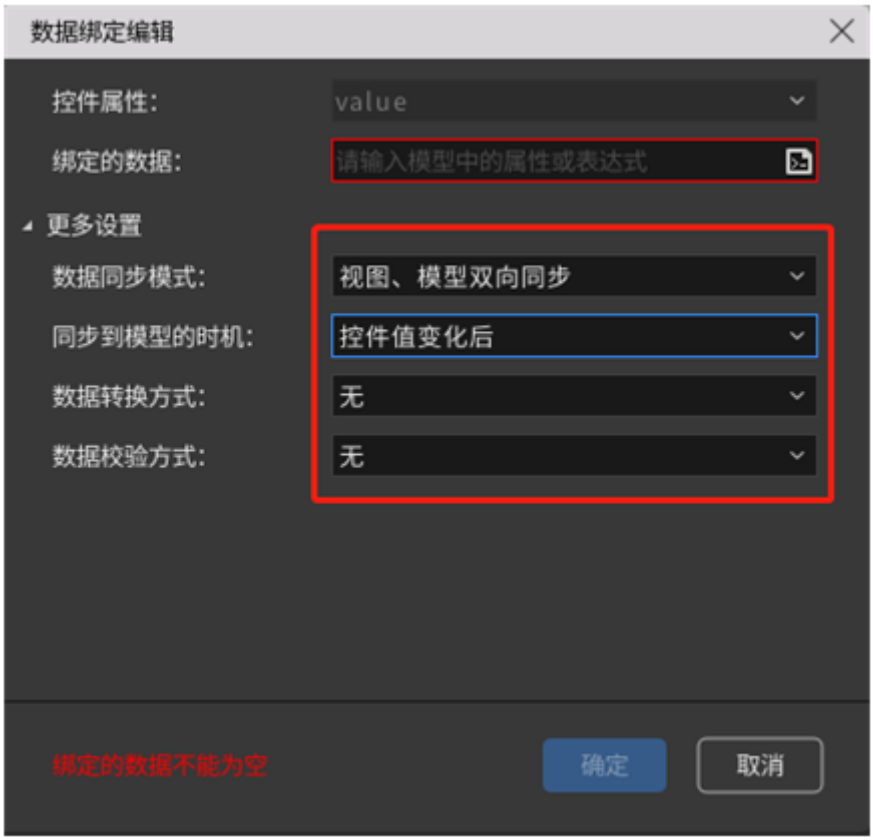

图 2.22　设置绑定规则

图 2.23　在按钮上停留了解使用方法

方式二：

步骤一：将 VM 上的属性（property 属性）用鼠标拖动到需要绑定的控件中，或者拖动到需要绑定的控件的某个属性中：

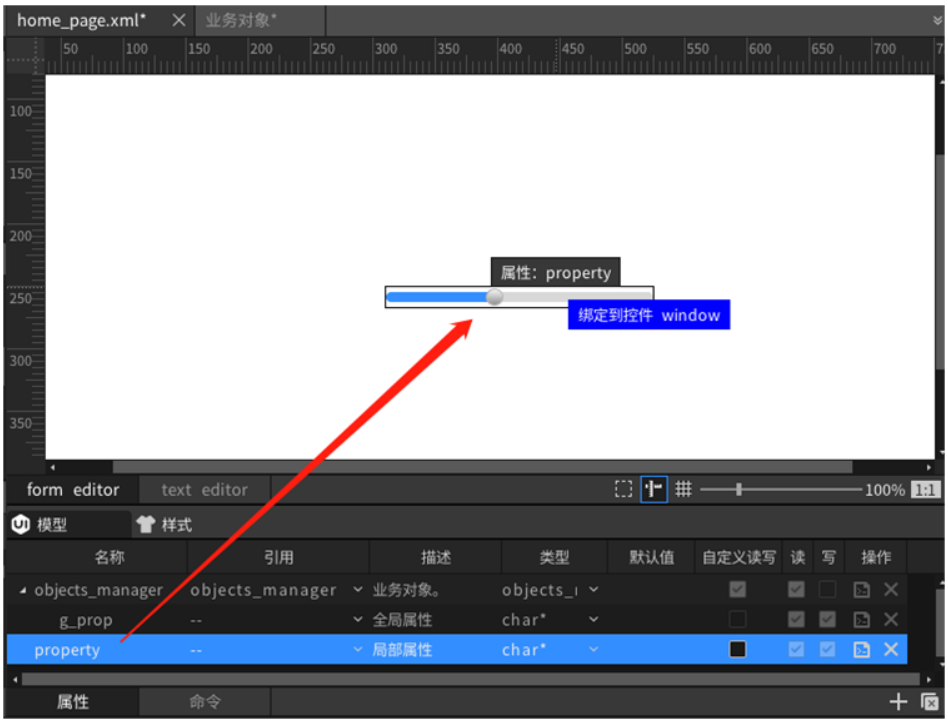

图 2.24　数据绑定步骤一

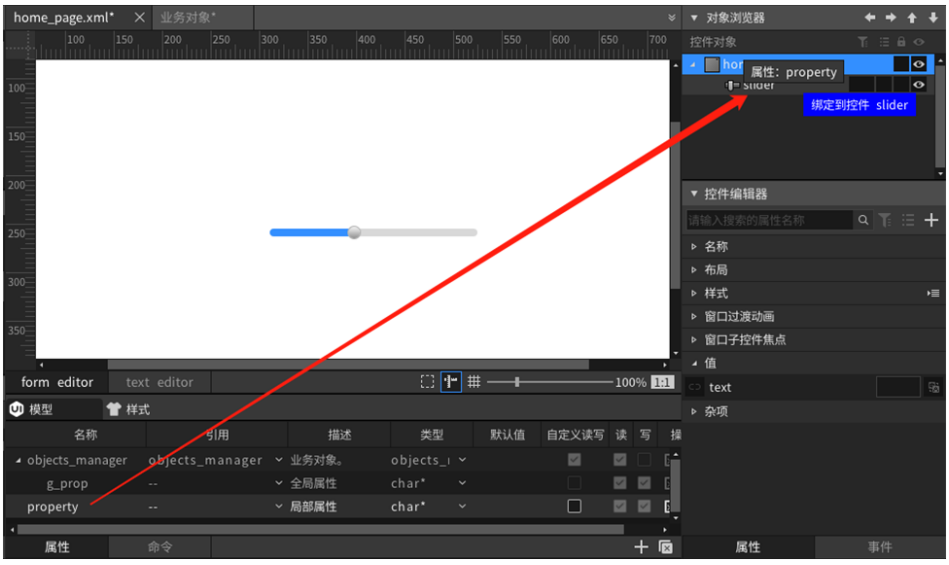

图 2.25　数据绑定步骤一

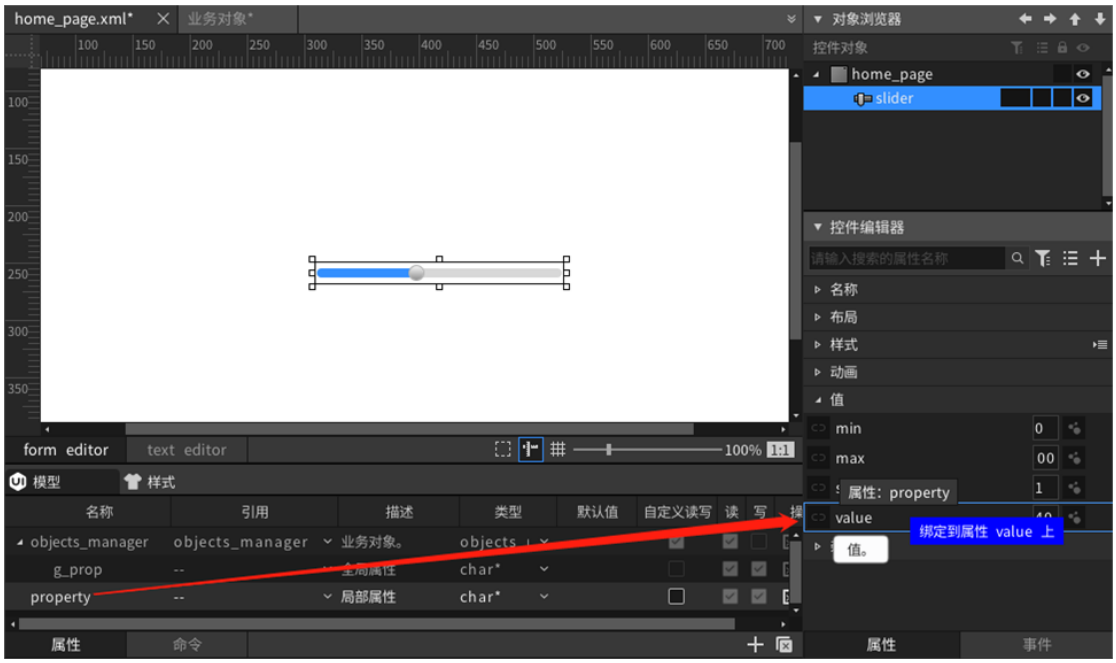

图 2.26　数据绑定步骤一

步骤二：弹出对话框，选择需要绑定的控件属性，点击绑定规则编辑框弹出绑定规则对话框，设置方式和方法一相同，设置完后点击确认即可完成绑定。

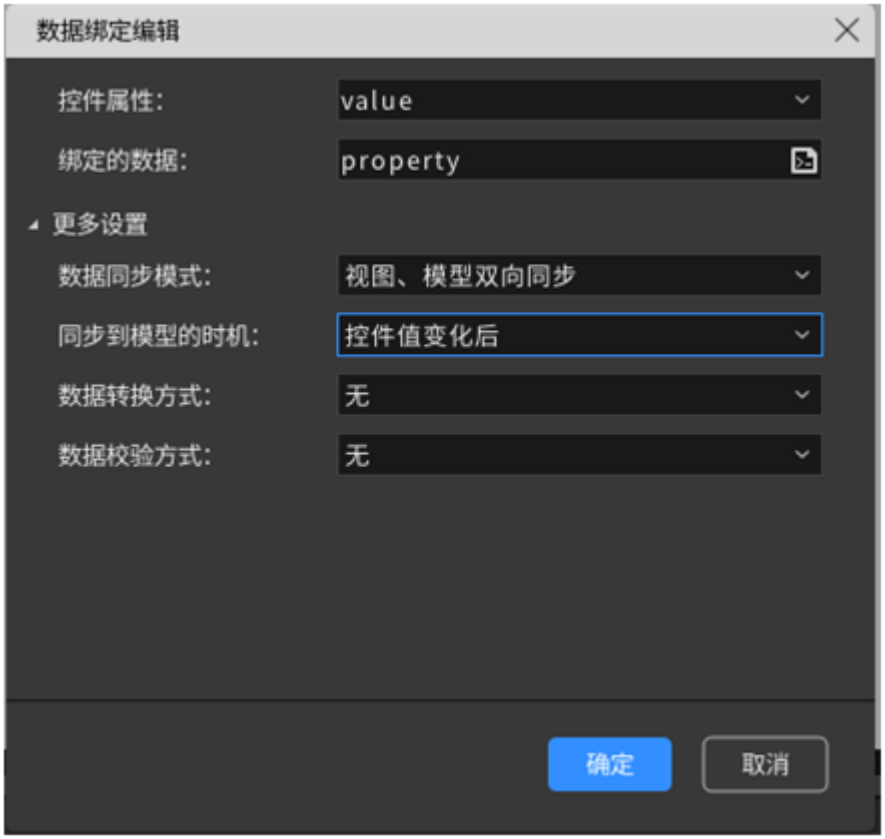

图 2.27　数据绑定步骤二

数据绑定完成后就会看到下图框选的按钮样式发生了改变：

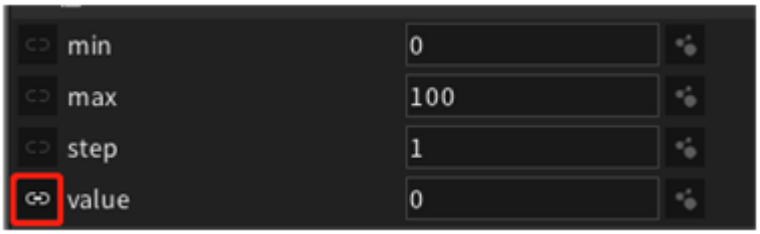

图 2.28　数据绑定完成

绑定成功后，再添加一个新的 label 控件，然后按照上面的方法把 property 属性绑定到 label 的 text 属性上，这样就可以在滑动 slider 滑块时，上面的 label 文本也会跟着改变。

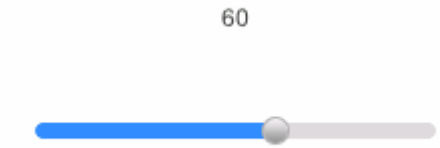

图 2.29　数据绑定运行结果

2.2.4　为 ViewModel 添加命令

虽然命令绑定不一定需要使用到 VM 中的命令，但是先了解如何为 VM 添加命令，后面就可以更全面地了解命令绑定了。命令分为局部命令和全局命令，局部命令只可以在该 VM 下使用，而全局命令可以在多个 VM 下共同使用（为 VM 添加命令后，对应的 VM 代码文件也会自动添加命令代码）。

1.　为 ViewModel 添加局部命令

现在为 home_page_view_model 添加一个局部命令。

步骤一：点击“模型”-> “命令”-> “+”按钮，之后就会在 VM 中增加一个局部命令：

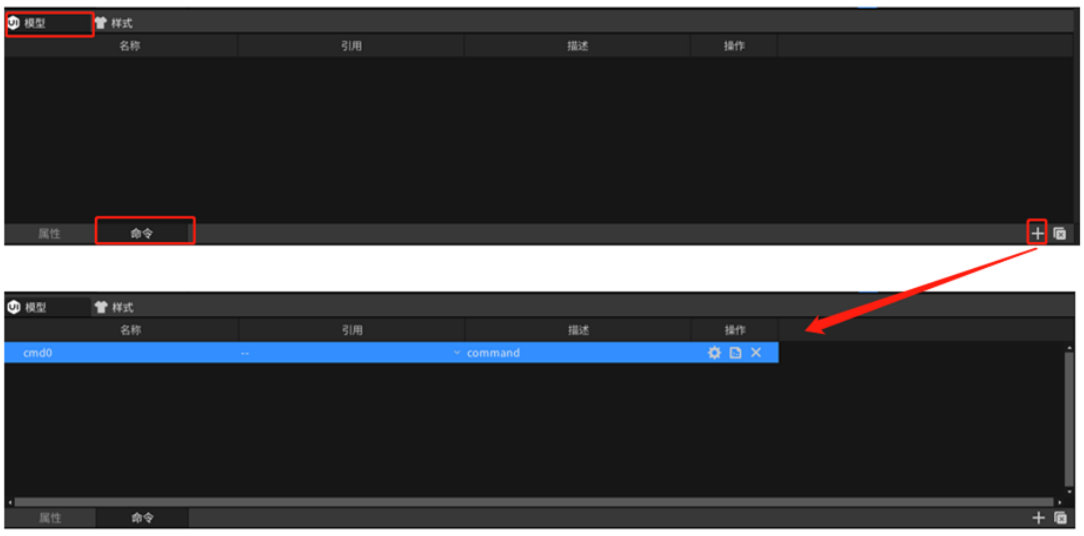

图 2.30　添加命令步骤一

步骤二：和编辑属性的操作一样，鼠标左键双击属性中的名称和描述可对其进行编辑：

图 2.31　编辑命令步骤二

（1）修改命令实现

步骤一：最后需要修改命令的实现，点击该命令的“跳转代码”按钮，打开对应 VM 代码文件：

图 2.32　编辑命令

步骤二：修改命令执行函数，如需要修改 home_page_view_model 的 command 命令的实现，就更改 home_page_view_model_command() 函数，如下图让命令打印语句”command run!\r\n”。home_page_view_model_can_command() 函数是判断命令能否执行的函数，在执行命令前会先执行该函数，返回 TRUE，则调用命令执行函数，否则不调用命令执行函数。：

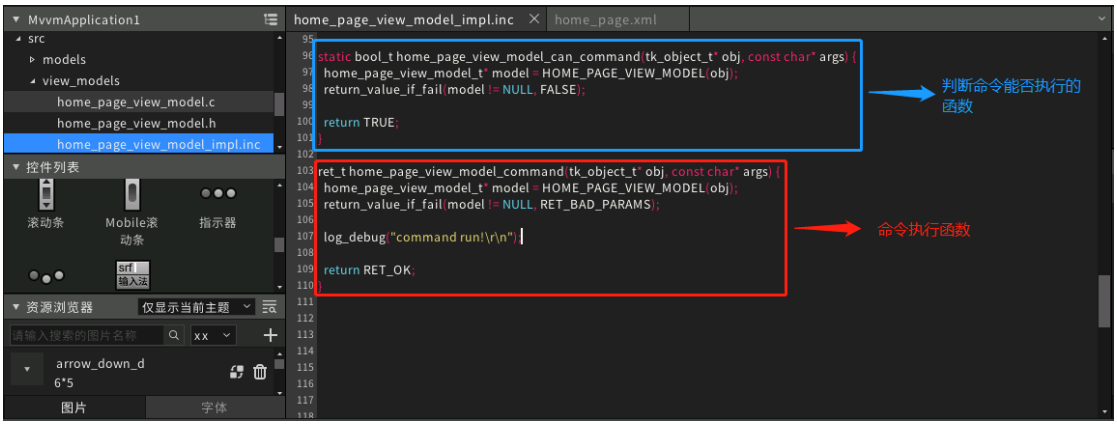

图 2.33　编辑命令代码

通过上述步骤，为 VM 添加局部命令就完成了。

2. 为 ViewModel 添加全局命令

（1）添加命令到全局对象管理器

步骤一：在菜单栏点击“查看”->“业务对象”按钮：

图 2.34 添加命令到全局对象管理器步骤一

步骤二：跳转到业务对象窗口，点击“命令”->“+”按钮，之后就会在全局对象管理器中增加一个命令，其代码中也会增加命令的实现函数：

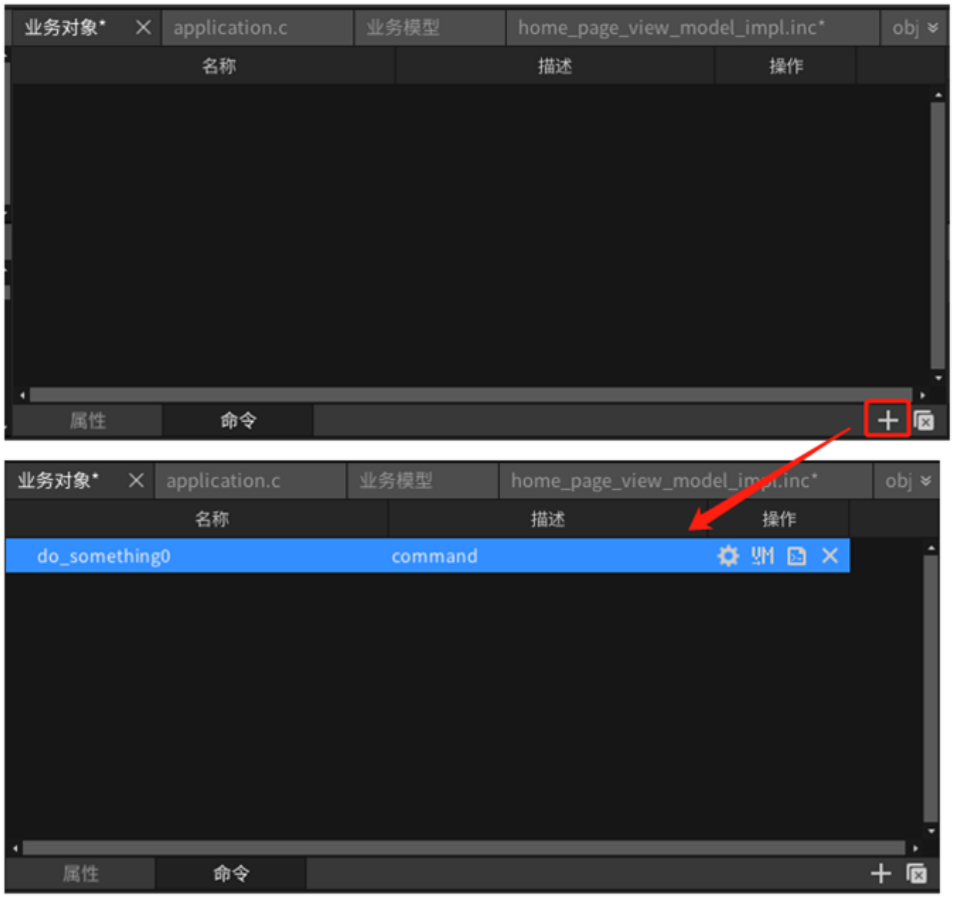

图 2.35 添加命令到全局对象管理器步骤二

步骤三：和编辑属性的操作一样，鼠标左键双击命令中的名称和描述可对其进行编辑：

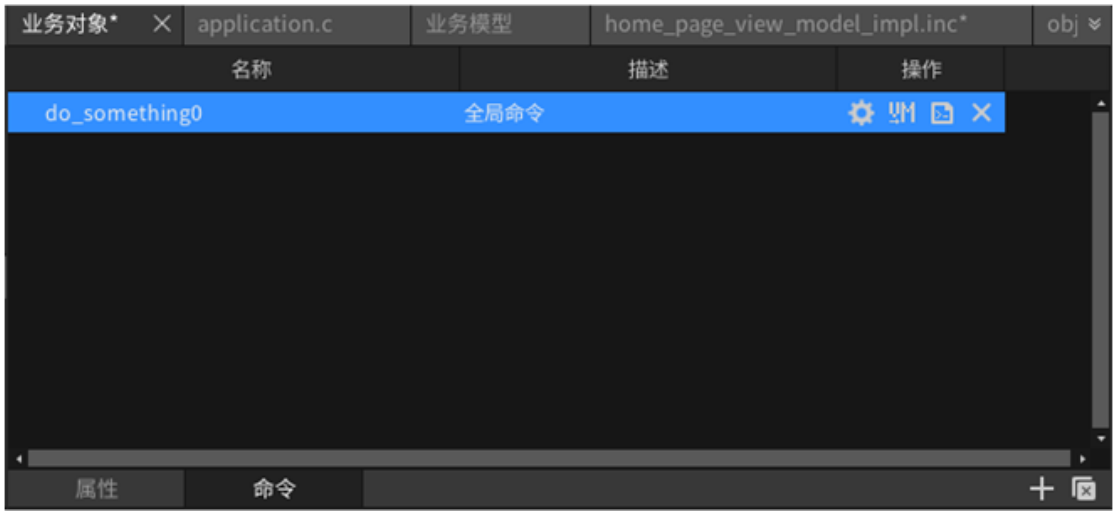

图 2.36　添加命令到全局对象管理器步骤三

步骤四：最后需要修改命令的实现，由于上文有详细解析，这里就不再赘述：

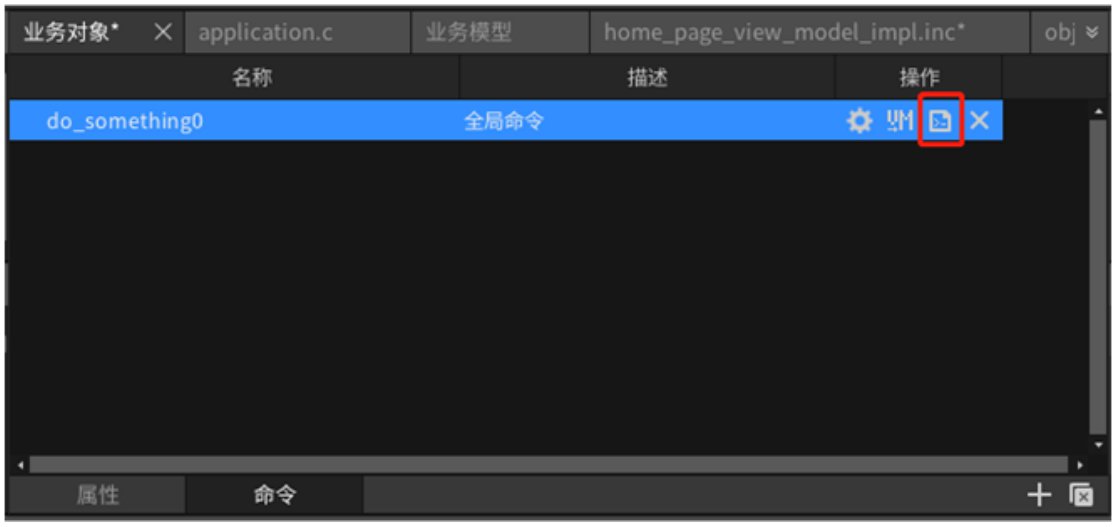

图 2.37　添加命令到全局对象管理器步骤四

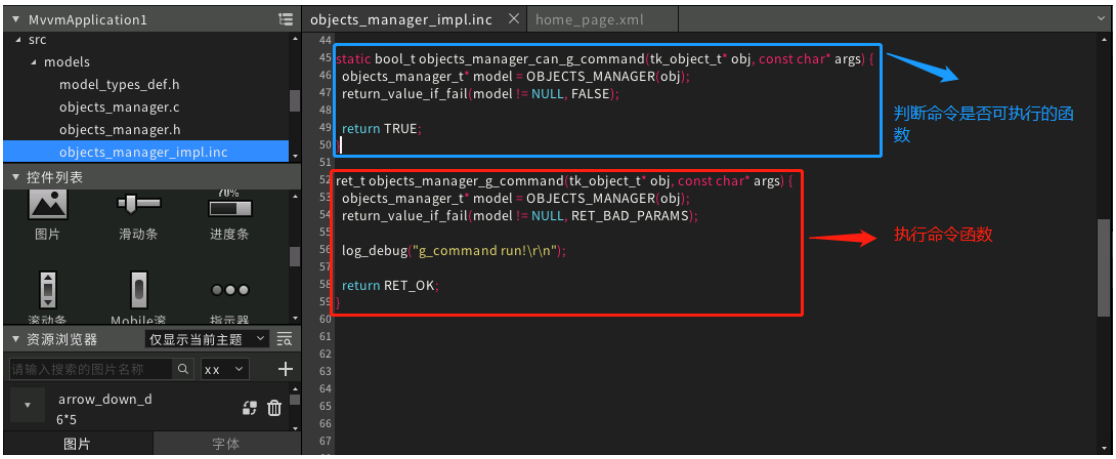

图 2.38　编辑全局命令代码

（2）添加全局命令到 ViewModel

回到 home_page 界面，就会发现多出一个命令 object_manager.g_command（：

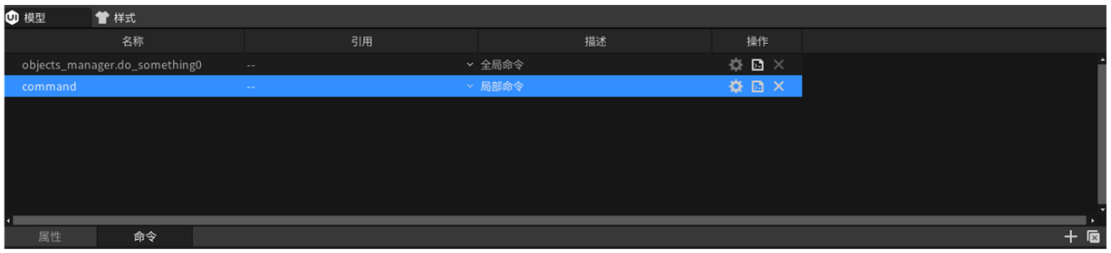

图 2.39　添加全局命令到视图模型

通过上述步骤，为 VM 添加全局命令就完成了。

2.2.5　命令绑定

下面讲解如何完成命令绑定。

1.　增加命令绑定规则

步骤一：选择需要绑定命令的控件，然后点击“事件”按钮，点击“+”按钮：

图 2.40　增加命令绑定规则步骤一

步骤二：之后会弹出对话框选择触发事件（该事件触发时，就会执行绑定的命令），如选择“value_changed”进入下一步，就会生成与控件 value_changed 事件绑定的命令：

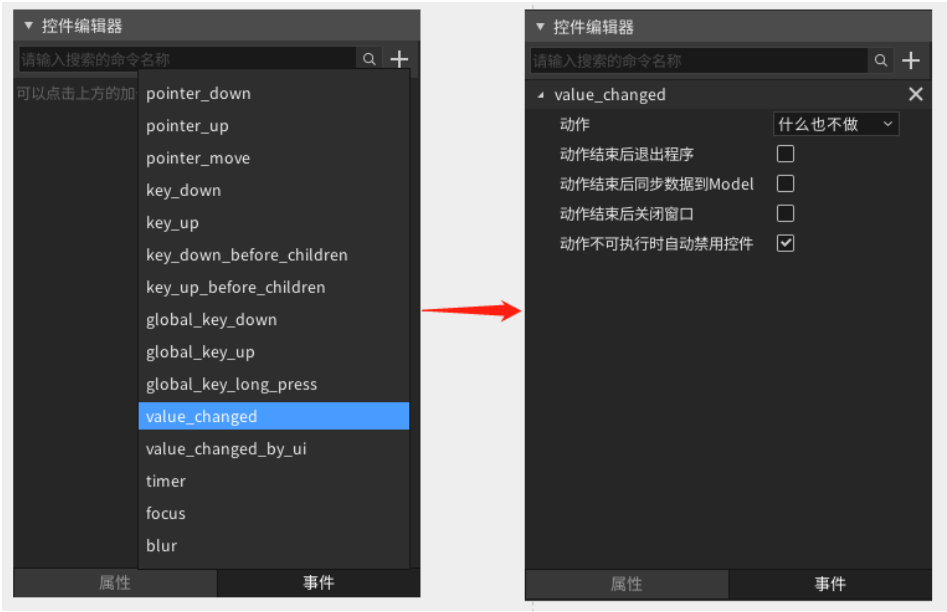

图 2.41　增加命令绑定规则步骤二

2.　选择执行动作

点击动作下拉列表，可以看到有如下动作：

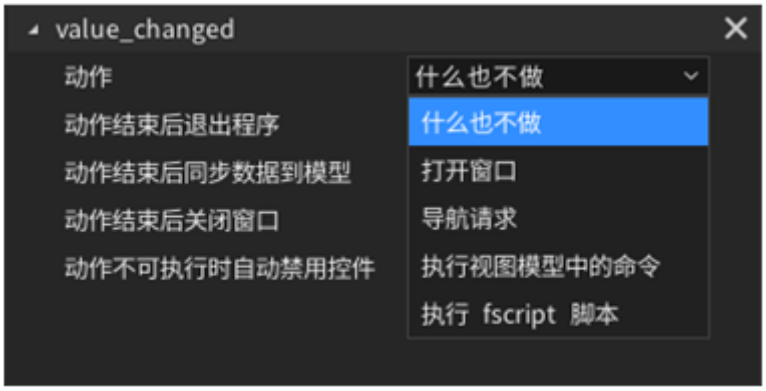

图 2.42　选择执行动作

每个动作的可选参数如下：

打开窗口：

可选参数	描述
窗口名称	指定打开窗口的名称
切换到已存在的窗口	存在同名窗口，会切换到已存在的窗口，而不会打开一个新窗口

导航请求：

可选参数	描述
请求类型	指定请求命令的名称
请求参数	请求命令需要传入的参数

执行 ViewModel 中的命令：

可选参数	描述
命令的名称	指定执行命令的名称
命令的参数	执行命令时需要传入的参数

执行 FScript 脚本：

可选参数	描述
FScript 代码	需要执行的 FScript 代码

下文逐一介绍（什么也不做顾名思义就是什么也不做，所以不作详细介绍）。

（1）打开窗口

用于打开指定窗口，例如窗口名称选择窗口 new，就会在 slider 滑动改变值后打开窗口 new：

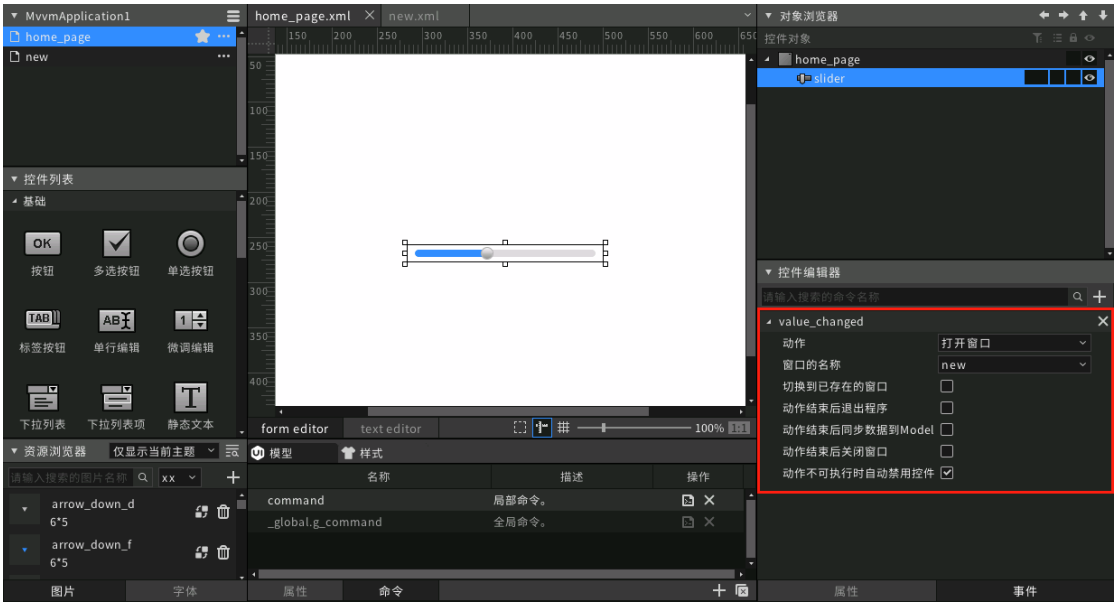

图 2.43　打开窗口

（2）导航请求

对导航器发起请求，请求后是否执行该动作由导航器进行判断，想深入了解可以查看 awtk-mvvm 源码里的 `docs/navigate.md` 文档

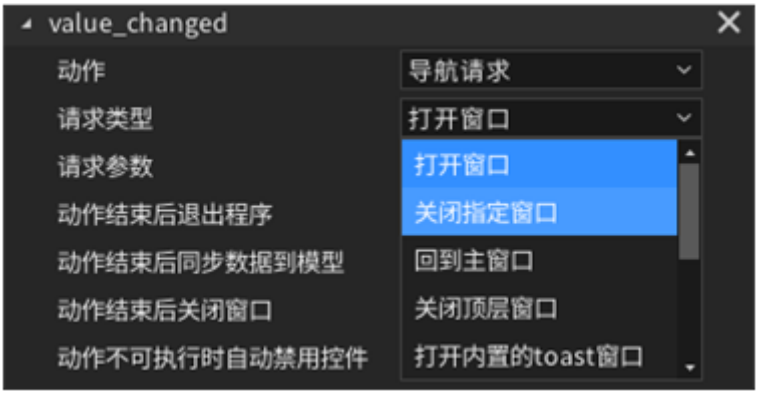

图 2.44　导航请求

图 2.45　导航请求参数

（3）执行 ViewModel 中的命令

用于执行在 VM 添加的命令，例如选择执行 command 命令（该命令是在上文添加的，命令实现为打印语句”command run!\r\n”）。

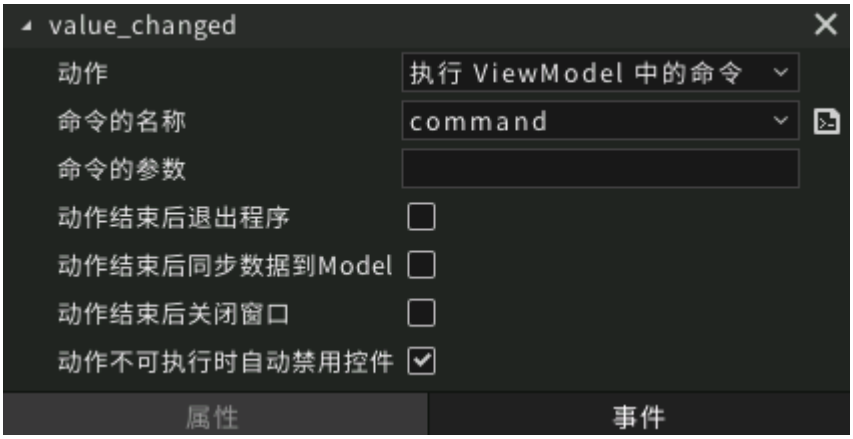

图 2.46　执行 VM 中的命令

在 slider 滑动改变值后执行命令打印该语句：

图 2.47　执行结果

指定命令的参数

可以指定一个参数，该参数并不是必须的，但有时命令参数确实能带来不少便利。

步骤一：点击命令的参数编辑框：

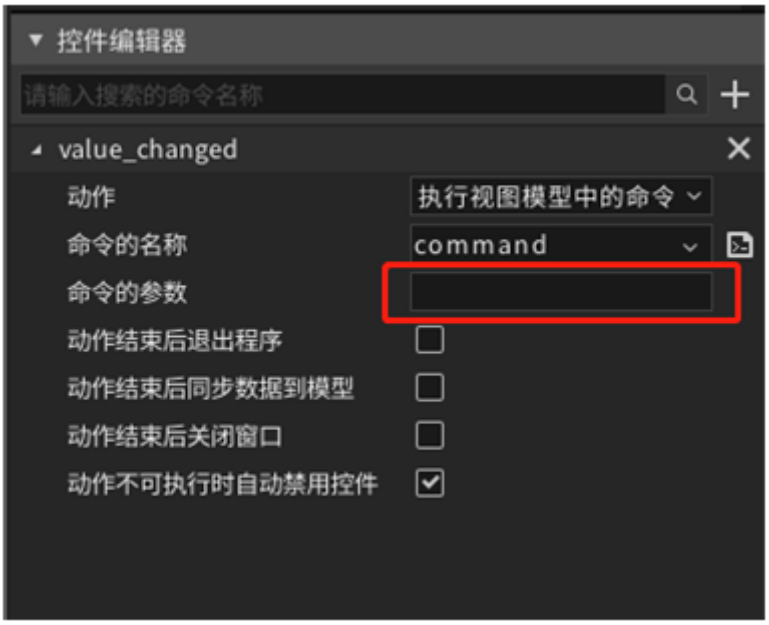

图 2.48　指定命令的参数步骤一

步骤二：弹出设置参数的对话框：

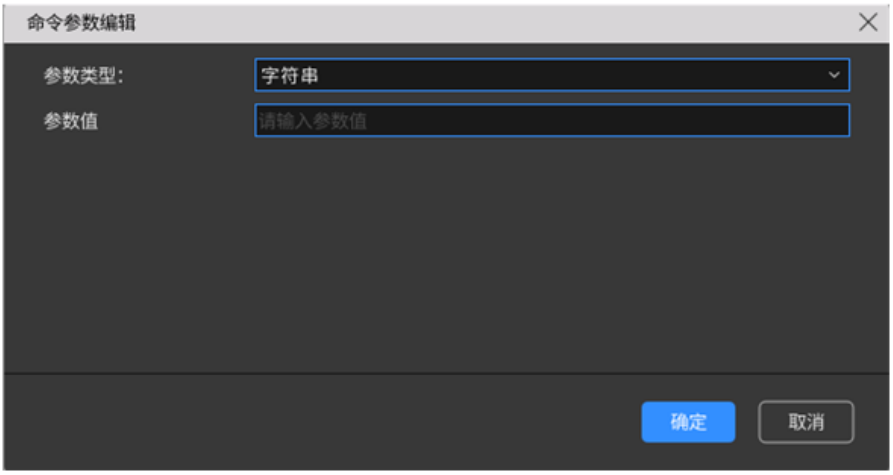

图 2.49　指定命令的参数步骤二

参数类型有三种：normal（单个字符串）、string（字符串形式的参数序列）和 fscript（FScript 表达式形式的参数序列），一般来说使用 normal 类型就足够了，如果想使用另外两种参数类型，可以选择对应类型，就可以在列表中填写所需要的数据（关于命令参数的更多资料请参阅 AWTK-MVVM 命令绑定[11]11.2 命令的参数）：

[11] https://github.com/zlgopen/awtk-mvvm/blob/master/docs/11.command_binding.md

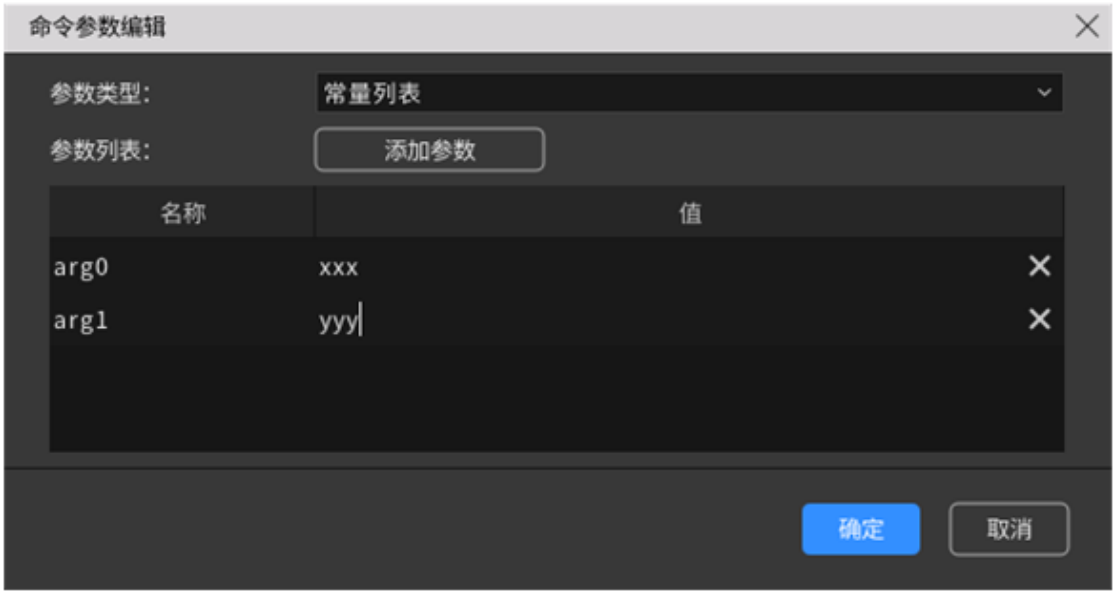

图 2.50　指定命令的参数步骤三

命令的参数会以字符串的形式传递到实现代码中：

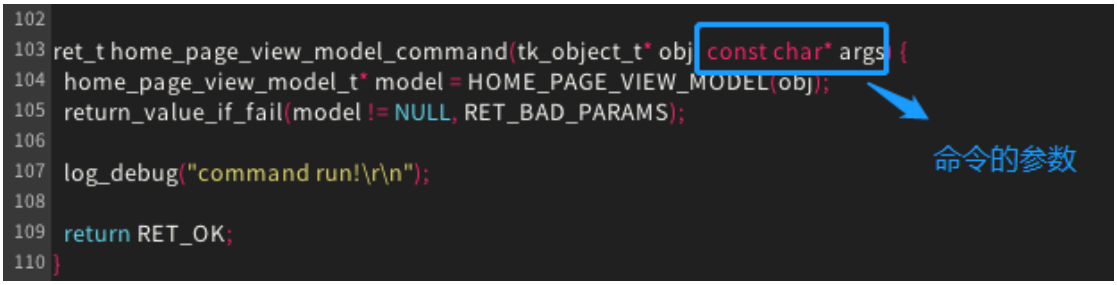

图 2.51　指定命令的参数步骤四

（4）执行 FScript 脚本

如果上述的动作都满足不了需求，则可以使用 FScript 脚本。

注: FScript 是一个极简的脚本引擎，借鉴了函数语言中一些思路，主要用于低端嵌入式系统，让用户轻松扩展现有系统，而不需要重新编译和下载固件。

步骤一：点击 FScript 代码编辑框：

图 2.52　执行 FScript 脚本步骤一

步骤二：在设定绑定规则对话框中的编辑器中编写 FScript 脚本：

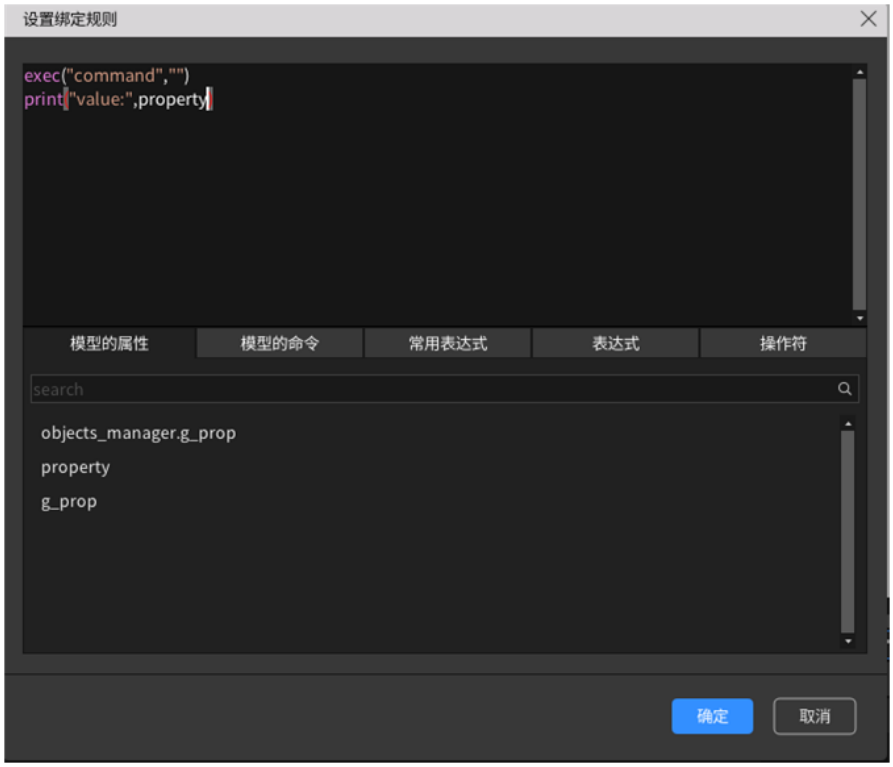

图 2.53　执行 FScript 脚本步骤二

关于 FScript 的更多资料请参阅 FScript[12]。

在 slider 滑动改变值后执行结果：

图 2.54　命令的参数

[12] https://github.com/zlgopen/awtk/blob/master/docs/fscript.md

3. 动作结束后行为

动作结束后行为在 AWTK-MVVM 命令绑定 [13]里有详细解释，在这里就不赘述了。

2.3 Model（模型）

如果只是做一个简单的应用程序，上面讲解的内容（VVM 模式）是完全够用的，但如果制作的应用程序需要在 VM 中添加大量的属性和命令，这样后面就很难对其进行维护和拓展。这时就需要用到 Model（模型，简称 M），模型相当于一个类，是对现实世界中业务逻辑的抽象，里面封装了其属性和行为。将 VM 中添加大量的属性和命令封装为一个个模型，最后再将模型对象添加为 VM 的属性，这样就是 MVVM 模式，解决了程序很难对其进行维护和拓展的问题。

2.3.1 新建 Model

步骤一：在菜单栏点击“查看”->“业务模型”按钮：

图 2.55 新建模型步骤一

步骤二：跳转到业务模型窗口后，点击下图框选按钮：

图 2.56 新建模型步骤二

步骤三：之后就会新建一个 model，对应的代码也会一并生成：

图 2.57 新建模型步骤三

[13] https://github.com/zlgopen/awtk-mvvm/blob/master/docs/11.command_binding.md

步骤四：双击模型中的名称和描述对其进行编辑：

图 2.58　新建模型步骤四

2.3.2　为 Model 添加属性

步骤一：选中模型，点击“+”按钮，之后就会在 M 中增加一个属性：

图 2.59　为模型添加属性步骤一

步骤二：编辑刚才在 M 中新增属性（和在 VM 编辑属性的操作步骤一样，这里不再赘述）：

图 2.60　为模型添加属性步骤二

2.3.3 为 Model 添加命令

步骤一：点击“命令”按钮，选中模型，点击“+”按钮，之后就会在 M 中增加一个命令：

图 2.61 为模型添加命令步骤一

步骤二：编辑刚才在 M 中新增命令：

图 2.62 为模型添加命令步骤二

步骤三：最后需要修改命令的实现，由于上文“为 ViewModel 添加命令”有详细解析，这里就不再赘述：

图 2.63 为模型添加命令步骤三

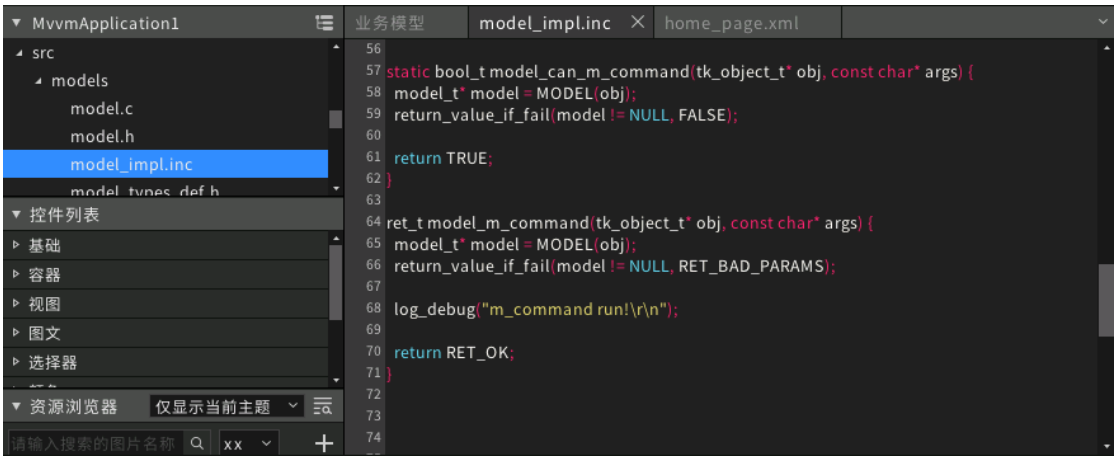

图 2.64　修改命令代码

2.3.4　添加 Model 对象到 ViewModel

和“为 ViewModel 添加属性”操作一样，只需把类型改为上文新建的模型即可：

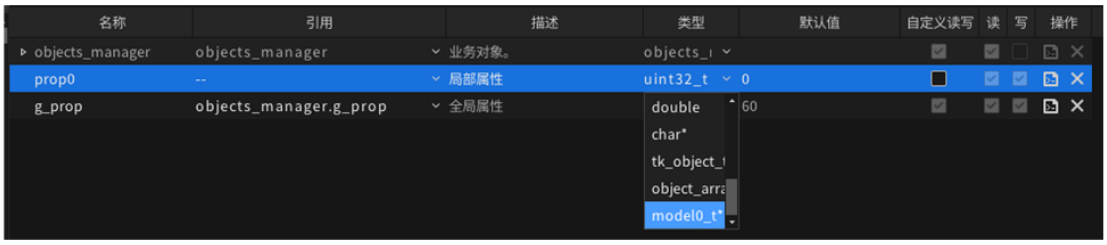

图 2.65　添加模型对象到视图模型

图 2.66　添加模型对象到视图模型

将上文绑定在 slider 和 label 的属性上，换成刚添加到 VM 里的 model 对象中的 m_prop 属性（数据绑定的方法在上文有详细教程）：

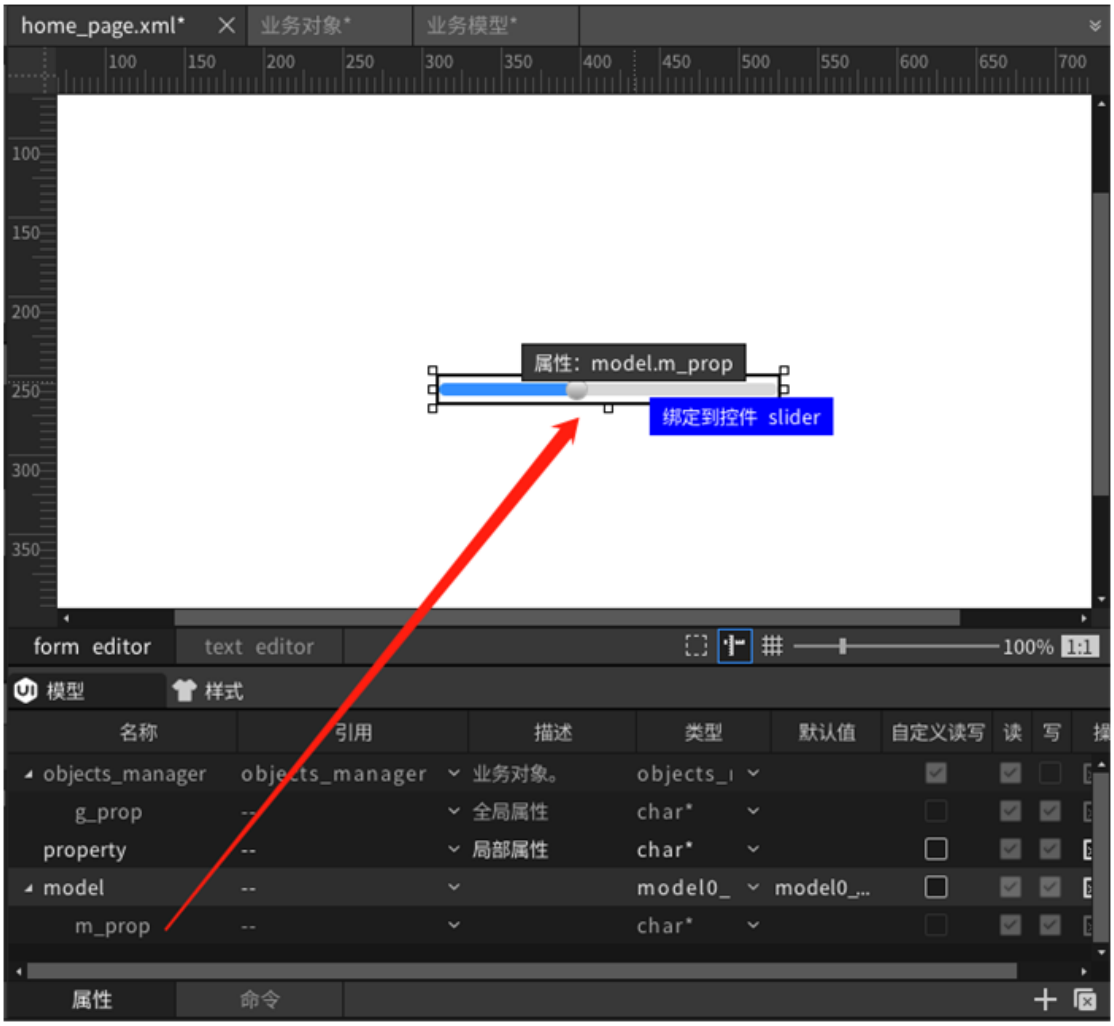

图 2.67　数据绑定

运行后拖动 slider 滑块得到的效果也是完全一样的：

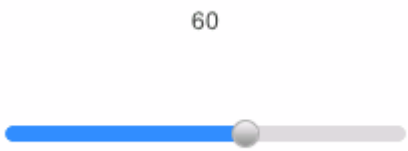

图 2.68　数据绑定运行结果

将上文绑定在 slider 的 value_changed 事件的 command 命令，换成刚添加到 VM 里的 model 对象中的 m_command 命令（命令绑定的方法在上文有详细教程）：

图 2.69　命令绑定

运行后拖动 slider 滑块得到的效果：

图 2.70　命令绑定运行结果

2.4　读写外部数据

如果在开发的应用程序需要将外部数据（如外部设备数据、网络数据）更新到界面上，或者将界面上的数据上传，那么下文将对项目的制作有所帮助。

2.4.1　同步读写模式

同步读写模式是程序需要读写外部数据时，UI 线程会阻塞，缺点是如果读写一次的时间久，那么界面会卡顿。优点是代码简单，容易维护且不容易出 Bug 。

首先需要为 model 添加命令，如在上一章 3 Model（模型）的 model 对象中的增加 read 和 write 两个命令：

图 2.71　增加读写命令

跳转到代码处，编写实现读写外部数据代码，下面的代码实现可供参考，使用命令参数 arg 判断模型对象读写其中某个属性，如果 arg 为空或为""，则整个模型对象进行读写：

```
ret_t model0_read(tk_object_t* obj, const char* args) {
  model0_t* model = MODEL0(obj);
  return_value_if_fail(model != NULL, RET_BAD_PARAMS);

    if(args == NULL || tk_str_eq(args,"")) {
        log_debug("读取外部数据所有内容到model\r\n");
    } else if(tk_str_eq(args,"m_prop")) {
        log_debug("读取外部数据xxx到model->m_prop\r\n");
    }

  return RET_OK;
}

ret_t model0_write(tk_object_t* obj, const char* args) {
  model0_t* model = MODEL0(obj);
  return_value_if_fail(model != NULL, RET_BAD_PARAMS);

    if(args == NULL || tk_str_eq(args,"")) {
        log_debug("将model所有内容上传");
    } else if(tk_str_eq(args,"m_prop")) {
        log_debug("上传model->m_prop\r\n");
    }

  return RET_OK;
}
```

（1）触发读写

触发读写是当用户每操作界面一次（如鼠标点击按钮），就读写一次。实现触发读写只需将上文添加的命令绑定到界面上的某个控件的某个事件即可，如需要点击 read 按钮更新 model 对象的 m_prop 属性，那么将 read 命令绑定到 read 按钮的 click 事件，命令参数设置为 m_prop 即可：

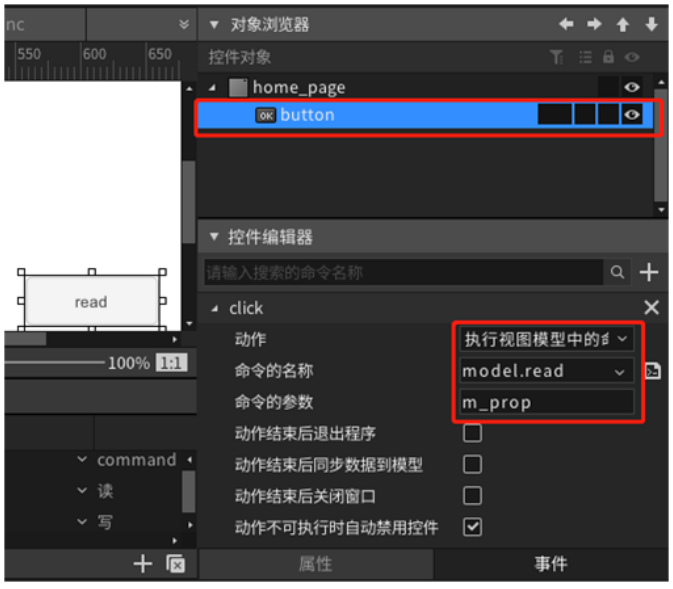

图 2.72 绑定同步触发读命令

运行后点击 read 按钮得到的效果：

图 2.73　绑定同步触发读命令结果

同理 write 命令的绑定也是一样，不再赘述。

（2）定时读写

定时读写是根据时间间隔进行周期性读写。实现定时读写首先需要为窗口绑定命令，在打开窗口时打开定时器（其参数为定时时间间隔，单位毫秒），在关闭窗口时关闭定时器。（想了解更多关于 FScript 定时器相关的函数，参阅 fscript 的 widget 扩展函数[14]）。

然后将读写命令绑定到窗口的 timer 事件上即可，这里只展示绑定 read 命令，write 命令的绑定也是一样的，不再赘述：

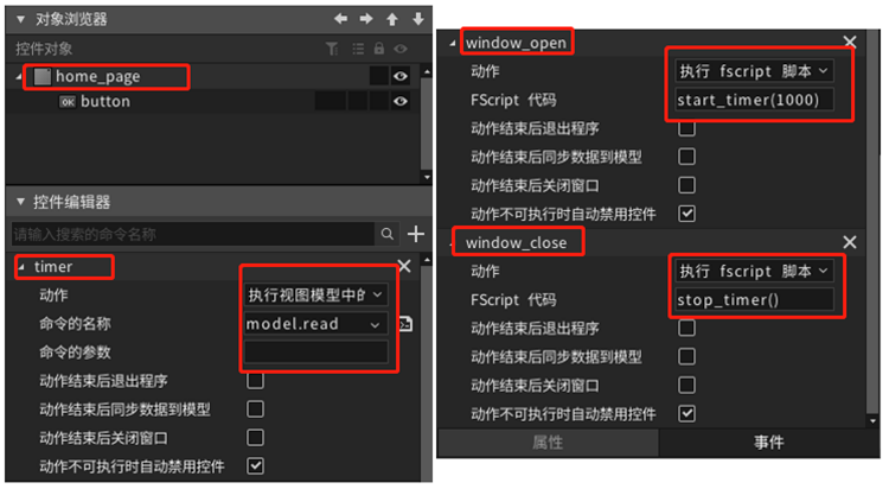

图 2.74　绑定同步定时读命令

运行后得到的结果：

图 2.75　绑定同步定时读命令结果

[14] https://github.com/zlgopen/awtk/blob/master/docs/fscript_widget.md

2.4.2 异步读写模式

异步读写模式相较于同步读写模式解决了在读写外部数据时，UI 线程会阻塞的问题，但同时代码也会变得更复杂，由于代码较为复杂，下文将由伪代码的形式讲解如何实现。

首先需要 application.c 处在程序运行增加读写线程（如不知该文件位置，可以看上文项目的目录结构），然后在函数内实现读写逻辑即可。（临界资源会在 UI 线程和读写数据线程中进行读写，所以需要互斥锁保证线程间不会同时对临界资源进行读写）：

```
/* application.c */

static ret_t application_on_launch(void) {
  objects_manager_set(objects_manager_create());

  /* 创建临界资源并创建互斥锁 */
  /* 创建读写数据线程并启动 */

  return RET_OK;
}
```

读写线程内部实现

读写线程内部实现按照下图实现即可：

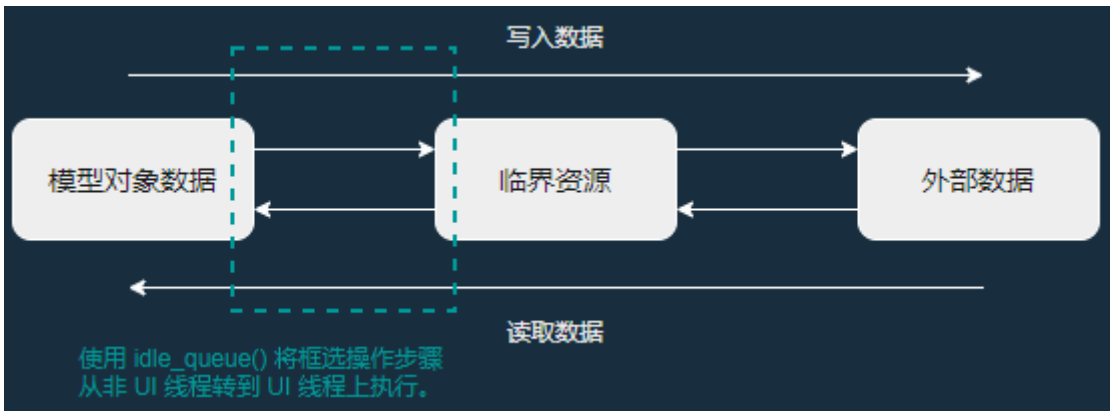

图 2.76　线程读写逻辑图

要注意的是模型对象数据和临界资源的数据交换需要放到 UI 线程里执行，因为如果放在非 UI 线程执行该操作时，用户操作界面时也会在 UI 线程中读写模型对象数据，这样就会让两条线程同时操作同一份数据，这种操作是不被允许的。idle_queue() 可以用于非 GUI 线程增加一个 idle，向 UI 线程主循环的事件队列中发送一个增加 idle 的请求，在 UI 线程里下一帧执行 idle 的回调函数。实现上图中的逻辑后，还可以在使用 idle_queue() 完成需补充的 UI 逻辑。

以下是读线程读取数据的伪代码：

```
/* application.c */

static bool_t g_is_quit_thread = FALSE; /* 关闭线程标志位 */
model_t g_tmp_model;                    /* 临界资源 */
tk_mutex_t* g_tmp_model_mutex = NULL;   /* 临界资源互斥锁 */
```

```c
/* 通知界面更新 */
static ret_t update_view(const idle_info_t* idle) {
  tk_object_t* model = TK_OBJECT(idle->ctx);

  /* 为 model 分发属性改变事件，通知 UI 界面更新 */
  emitter_dispatch_simple_event(&model->emitter, EVT_PROPS_CHANGED);

  return RET_OK;
}

/* 弹出警告对话框 */
static ret_t pop_warn_dialog(const idle_info_t* idle) {
  const char* log = (const char*)(idle->ctx);

  dialog_warn(NULL, log);

  return RET_OK;
}

/* 模型对象读取临界资源数据 */
static ret_t read_tmp_data(const idle_info_t* idle) {
  model_t* model = (model_t*)(idle->ctx);

  tk_mutex_lock(g_tmp_model_mutex);
  /* model 读取临界资源数据，调用 update_view() 函数可将 model 数据更新到 UI 界面上
↪*/
  /* model->m_prop = g_tmp_model.m_prop */
  tk_mutex_unlock(g_tmp_model_mutex);

  return RET_OK;
}

/* 临界资源读取外部数据 */
static ret_t read_external_data(void) {
  tk_mutex_lock(g_tmp_model_mutex);
  /* 临界资源读取外部数据（如网络数据、本地文件数据） */
  /* g_tmp_model.m_prop = xxx */
  tk_mutex_unlock(g_tmp_model_mutex);

  return RET_OK;
}

/* 读线程 */
static void* read_thread_entry(void* args) {
  ret_t ret = RET_SKIP;
  objects_manager_t* om = OBJECTS_MANAGER(objects_manager());
  tk_object_t* model = om->model; /* 模型对象 */
```

```
  while (!g_is_quit_thread) {
    /* 临界资源读取外部数据 */
    ret = read_external_data();

    if (ret == RET_OK) {
      /* 临界资源读取外部数据成功时，让模型对象读取临界资源，并通知界面更新 */
      idle_queue(read_tmp_data, model);
      idle_queue(update_view, model);
    } else {
      /* 临界资源读取外部数据失败时，弹出对话框通知 */
      idle_queue(pop_warn_dialog, "Read Fail!");
    }
    sleep_ms(100);
  }

  return NULL;
}
```

以下是写线程写入数据的伪代码：

```
/* application.c */

bool_t g_write_data_req = FALSE; /* 写数据请求 */

/* 临界资源写入外部数据 */
static ret_t write_external_data(void) {
  ret_t ret = RET_SKIP;
  tk_mutex_lock(g_tmp_model_mutex);
  if (g_write_data_req) { /* 是否有写数据请求，如没有则跳过 */
    /* 临界资源写入外部数据，如将临界资源数据上传到服务器上 */
    /* xxx = g_tmp_model.m_prop */
    g_write_data_req = FALSE;  /* 完成写数据请求 */
    ret = RET_OK;
  }
  tk_mutex_unlock(g_tmp_model_mutex);

  return ret;
}

/* 写线程 */
static void* write_thread_entry(void* args) {
  ret_t ret = RET_SKIP;
  objects_manager_t* om = OBJECTS_MANAGER(objects_manager());
  tk_object_t* model = om->model; /* 模型对象 */

  while (!g_is_quit_thread) {
    /* 临界资源写入外部数据 */
    ret = write_external_data();
    if (ret != RET_SKIP) {
      /* 通知 UI 写入成功或失败 */
```

```
      if (ret == RET_OK) {
        idle_queue(pop_warn_dialog, "upload Succeed!");
      } else {
        idle_queue(pop_warn_dialog, "upload Fail!");
      }
    }
    sleep_ms(100);
  }
  return NULL;
}
```

在 model 增加 write 命令，增加 write 命令可以参考上文同步读写模式。

```
/* model_impl.inc */

extern model_t g_tmp_model;
extern tk_mutex_t* g_tmp_model_mutex;
extern bool_t g_write_data_req;

ret_t model_write(tk_object_t* obj, const char* args) {
  model_t* model = MODEL(obj);
  return_value_if_fail(model != NULL, RET_BAD_PARAMS);

  tk_mutex_lock(g_tmp_model_mutex);
  /* 将模型对象数据写入到临界资源中 */
  g_tmp_model.m_prop = tk_object_get_prop_uint32(obj, "m_prop", 0);

  g_write_data_req = TRUE; /* 请求将临界资源写入外部数据 */
  tk_mutex_unlock(g_tmp_model_mutex);

  return RET_OK;
}
```

3. MVVM-C 案例分析

上一章讲解了如何制作 MVVM-C 项目，为了加深大家对 MVVM-C 的理解，本章将以 Mvvm-C-Demo 为例进行 MVVM-C 项目的案例分析，该案例主要功能如下：

- 显示多个设备。
- 可以增加和删除设备。
- 解锁后可以修改设备类型和设备参数。
- 可以还原和清空设备列表。
- 不同设备类型显示不同数据，下图为案例项目中的设备参数表：

设备参数类型	
0	NONE
1	NTC_2552K
2	NTC_5K
3	NTC_10K

设备类型和数据对应表	
设备类型	数据
PACK_SKP_1000	(bool) IO1、IO2
PACK_SKP_2000	(double) temp
PACK_SKP_3042	(double) temp
PACK_SKP_3132	(int) A1、A2
PACK_SKP_3142	(int) A1、A2
PACK_SKP_5002	(double) TPS

图 3.1　案例项目中的设备参数表

界面及功能如图所示：

图 3.2　案例项目界面

3.1 模型设计

首先使用 AWTK Designer 新建一个 MVVM-C 项目工程，步骤详见上文 2.1 章节，然后判断该项目是否需要增加模型，根据上文案例项目的需要的功能，显然是需要增加一个设备模型，此处将该模型名为”device”，该模型包含设备类型、设备参数和数据，具体详见下表：

属性名称	类型	描述	备注
pack_type	device_type_t	设备类型	自定义枚举类型，影响设备数据
pack_params	device_params_t	设备参数	自定义枚举类型
io1	bool_t	设备数据	随机生成，由设备类型决定是否显示
io1	bool_t	设备数据	随机生成，由设备类型决定是否显示
a1	int32_t	设备数据	随机生成，由设备类型决定是否显示
a2	int32_t	设备数据	随机生成，由设备类型决定是否显示
temp	double	设备数据	随机生成，由设备类型决定是否显示
tps	double	设备数据	随机生成，由设备类型决定是否显示

注: 本案例中”device”模型的 pack_type 属性和 pack_params 属性均为自定义的枚举类型，实现方式详见下一小节，其中 pack_type 属性将决定该设备显示哪些数据，为了方便演示效果，这些设备数据均是随机生成的。

3.1.1 增加自定义枚举类型

为了代码拥有更好的可读性，可以将设备类型和设备参数自定义为一种枚举类型，打开项目目录下“src/models/model_types_def.h”，在该文件中新增自定义枚举类型（注意：注释不可删除并且需要按照格式写，否则在 AWTK Designer 下无法找到该枚举类型）。

设备类型：

- PACK_SKP_1000
- PACK_SKP_2000
- PACK_SKP_3042
- PACK_SKP_3132
- PACK_SKP_3142
- PACK_SKP_5002

在代码处增加设备类型的枚举类型：

```
/* src/models/model_types_def.h */

/**
 * @enum device_type_t
 * @prefix DEV_TYPE_
*/
typedef enum _device_type_t {
  /**
   * @const DEV_TYPE_PACK_SKP_1000
   */
  DEV_TYPE_PACK_SKP_1000 = 0,
  /**
```

```
   * @const DEV_TYPE_PACK_SKP_2000
   */
  DEV_TYPE_PACK_SKP_2000,
  /**
   * @const DEV_TYPE_PACK_SKP_3042
   */
  DEV_TYPE_PACK_SKP_3042,
  /**
   * @const DEV_TYPE_PACK_SKP_3132
   */
  DEV_TYPE_PACK_SKP_3132,
  /**
   * @const DEV_TYPE_PACK_SKP_3142
   */
  DEV_TYPE_PACK_SKP_3142,
  /**
   * @const DEV_TYPE_PACK_SKP_5002
   */
  DEV_TYPE_PACK_SKP_5002,

  DEV_TYPE_MAX_COUNT
} device_type_t;
```

设备参数类型：

- NONE
- NTC_2252K
- NTC_5K
- NTC_10K

在代码处增加设备参数类型的枚举类型：

```
/* src/models/model_types_def.h */

/**
 * @enum device_params_t
 * @prefix DEV_PARAMS_
*/
typedef enum _device_params_t {
  /**
   * @const DEV_PARAMS_NONE
   */
  DEV_PARAMS_NONE = 0,
  /**
   * @const DEV_PARAMS_NTC_2252K
   */
  DEV_PARAMS_NTC_2252K,
  /**
   * @const DEV_PARAMS_NTC_5K
   */
```

```
  DEV_PARAMS_NTC_5K,
  /**
   * @const DEV_PARAMS_NTC_10K
   */
  DEV_PARAMS_NTC_10K,

  DEV_PARAMS_MAX_COUNT
} device_params_t;
```

3.1.2 新增设备模型

最后按照案例的需求，新增“设备”模型：device，步骤详见上文 2.3.1 章节。

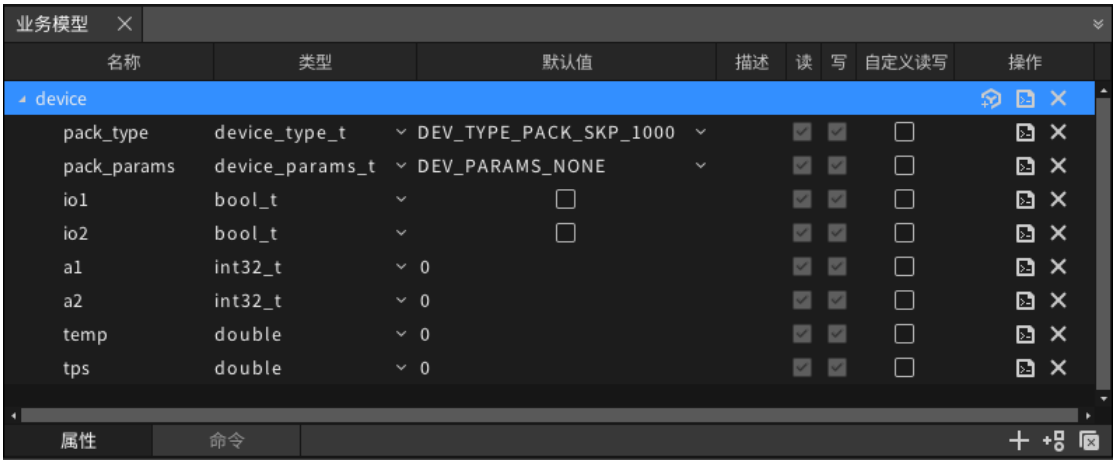

图 3.3 新增设备模型

在 AWTK Designer 新增 device 模型保存后，会自动在项目目录下“src/models”生成 device 模型的相关代码。

3.2 视图模型设计

接下来是为主界面 home_page 新建一个视图模型 home_page_view_model，步骤详见上文 2.2.1 章节，在 AWTK Designer 新增视图模型 home_page_view_model 保存后，会自动在项目目录下“src/view_models”生成 home_page_view_model 的相关代码。

3.2.1 为视图模型增加属性

根据案例项目的需要的功能，为视图模型增加下图属性：

图 3.4 为视图模型增加属性

items 对象数组和 current_index 属性在实现命令功能时就会使用，而 unlocked 属性会在讲解界面设计时才会使用。

3.2.2 为视图模型增加命令

根据案例项目的需要的功能，为视图模型增加下图命令：

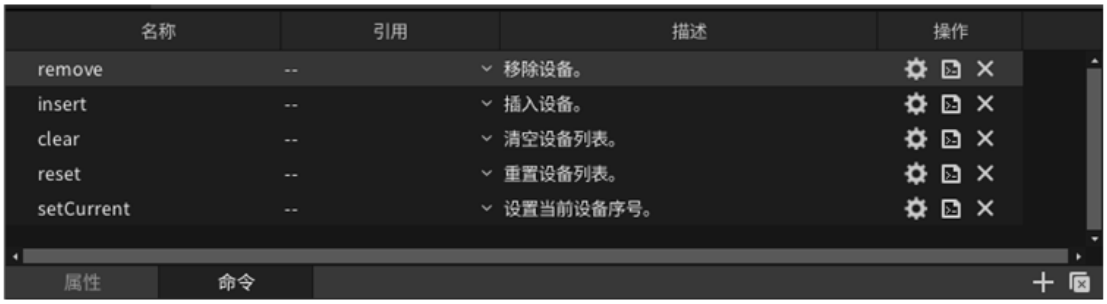

图 3.5 为视图模型增加命令

最后需要跳转到命令代码，实现命令功能函数和判断命令是否可执行的函数。

3.2.3 实现设置当前设备序号功能

该功能是用于设置设备插入或删除序号，是实现插入设备功能和删除设备功能的前提功能，通过执行 setCurrent 命令，更改 current_index 属性的值。

```
/* src/view_models/home_page_view_model_impl.inc */

static bool_t home_page_view_model_can_setCurrent(tk_object_t* obj, const char*␣
↪args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  (void)args;
  return_value_if_fail(model != NULL, FALSE);

  return TRUE;
}

ret_t home_page_view_model_setCurrent(tk_object_t* obj, const char* args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  tk_object_t* a = object_default_create();
  (void)args;
  return_value_if_fail(model != NULL && a != NULL, RET_BAD_PARAMS);

  /* 将字符串类型的命令参数转换为 object 对象 */
  tk_command_arguments_to_object(args, a);
  /* 更新当前选中设备序号 */
  model->current_index = tk_object_get_prop_int32(a, "index", -1);
  TK_OBJECT_UNREF(a);
  return RET_OBJECT_CHANGED;
}
```

3.2.4 实现插入设备功能

该功能是将新增设备插入到设备列表里当前选中设备序号处，通过执行 insert 命令，将新建的 device 设备对象，插入到 items 对象数组里的 current_index（该值通过设置当前设备序号功能修改）处。

```
/* src/view_models/home_page_view_model_impl.inc */

static bool_t home_page_view_model_can_insert(tk_object_t* obj, const char* args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  object_array_t* items = NULL;
  (void)args;
  return_value_if_fail(model != NULL, FALSE);

  items = OBJECT_ARRAY(model->items);
  return_value_if_fail(items != NULL, FALSE);

  /* 当前选中设备序号合法时，才可以使用 insert 命令 */
  return model->current_index >= 0 && model->current_index < items->size;
}

ret_t home_page_view_model_insert(tk_object_t* obj, const char* args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  value_t v;
  tk_object_t* device = NULL;
  ret_t ret = RET_OK;
  (void)args;
  return_value_if_fail(model != NULL && model->items != NULL, RET_BAD_PARAMS);

  /* 创建设备并插入到设备列表里当前选中设备序号处 */
  device = device_create();
  return_value_if_fail(device != NULL, RET_FAIL);

  value_set_object(&v, device);
  emitter_on(EMITTER(device), EVT_PROP_CHANGED, emitter_forward, model);
  ret = object_array_insert(model->items, model->current_index, &v);
  TK_OBJECT_UNREF(device);

  return ret;
}
```

3.2.5 实现移除设备功能

该功能是将设备列表里当前选中设备序号处的设备移除，通过执行 remove 命令，将 items 对象数组里的 current_index （该值通过设置当前设备序号功能修改）处 device 设备对象移除。

```
/* src/view_models/home_page_view_model_impl.inc */

static bool_t home_page_view_model_can_remove(tk_object_t* obj, const char* args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  object_array_t* items = NULL;
  (void)args;
  return_value_if_fail(model != NULL, FALSE);

  items = OBJECT_ARRAY(model->items);
  return_value_if_fail(items != NULL, FALSE);

  /* 当前选中设备序号合法时，才可以使用 remove 命令 */
  return model->current_index >= 0 && model->current_index < items->size;
}

ret_t home_page_view_model_remove(tk_object_t* obj, const char* args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  (void)args;
  return_value_if_fail(model != NULL && model->items != NULL, RET_BAD_PARAMS);

  /* 删除设备列表里当前选中设备序号处的设备 */
  return object_array_remove(model->items, model->current_index);
}
```

3.2.6 实现清除设备列表功能

该功能是将设备列表里的所有设备，通过执行 clear 命令，将 items 对象数组里的所有 device 设备对象移除。

```
/* src/view_models/home_page_view_model_impl.inc */

static bool_t home_page_view_model_can_clear(tk_object_t* obj, const char* args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  object_array_t* items = NULL;
  (void)args;
  return_value_if_fail(model != NULL, FALSE);

  items = OBJECT_ARRAY(model->items);
  return_value_if_fail(items != NULL, FALSE);

  /* 设备列表不为空时，才可以使用 clear 命令 */
  return items->size > 0;
}
```

```
ret_t home_page_view_model_clear(tk_object_t* obj, const char* args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  return_value_if_fail(model != NULL && model->items != NULL, RET_BAD_PARAMS);
  (void)args;

  /* 清空设备列表 */
  return object_array_clear_props(model->items);
}
```

3.2.7 实现重置设备列表功能

该功能是将重置设备列表，为了方便演示效果，该功能的实现是插入 50 个属性随机的设备对象到设备列表中，通过执行 reset 命令，在 items 对象数组里增加 50 个属性随机的 device 设备对象。

```
/* src/view_models/home_page_view_model_impl.inc */

static bool_t home_page_view_model_can_reset(tk_object_t* obj, const char* args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  object_array_t* items = NULL;
  return_value_if_fail(model != NULL, FALSE);

  items = OBJECT_ARRAY(model->items);
  return_value_if_fail(items != NULL, FALSE);

  /* 设备列表为空时，才可以使用 reset 命令 */
  return items->size == 0;
}

ret_t home_page_view_model_reset(tk_object_t* obj, const char* args) {
  home_page_view_model_t* model = HOME_PAGE_VIEW_MODEL(obj);
  ret_t ret = RET_OK;
  uint32_t i = 0;
  (void)args;
  return_value_if_fail(model != NULL && model->items != NULL, RET_BAD_PARAMS);

  /* 插入50个属性随机的设备对象到设备列表中 */
  for (i = 0; i < 50 && ret == RET_OK; i++) {
    tk_object_t* device = device_create();
    if (device != NULL) {
      value_t v;
      tk_object_set_prop_int(device, "pack_type", random() % DEV_TYPE_MAX_COUNT);
      tk_object_set_prop_int(device, "pack_params", random() % DEV_PARAMS_MAX_
↪COUNT);
      tk_object_set_prop_bool(device, "io1", random() % 2 == 0);
      tk_object_set_prop_bool(device, "io2", random() % 2 == 0);
      tk_object_set_prop_double(device, "temp", random() / 10.0);
      tk_object_set_prop_int32(device, "a1", random());
      tk_object_set_prop_int32(device, "a2", random());
```

```
      tk_object_set_prop_double(device, "tps", random() / 10.0);

      value_set_object(&v, device);
      object_array_push(model->items, &v);
      emitter_on(EMITTER(device), EVT_PROP_CHANGED, emitter_forward, model);
      TK_OBJECT_UNREF(device);
    } else {
      ret = RET_FAIL;
    }
  }

  return ret;
}
```

3.3 界面设计

最后是界面设计，在案例项目里需要显示多个设备，使用列表视图是一个不错的选择，按照《AWTK-Designer 快速使用指南》进行界面设计，可以得到设备列表界面的雏形：

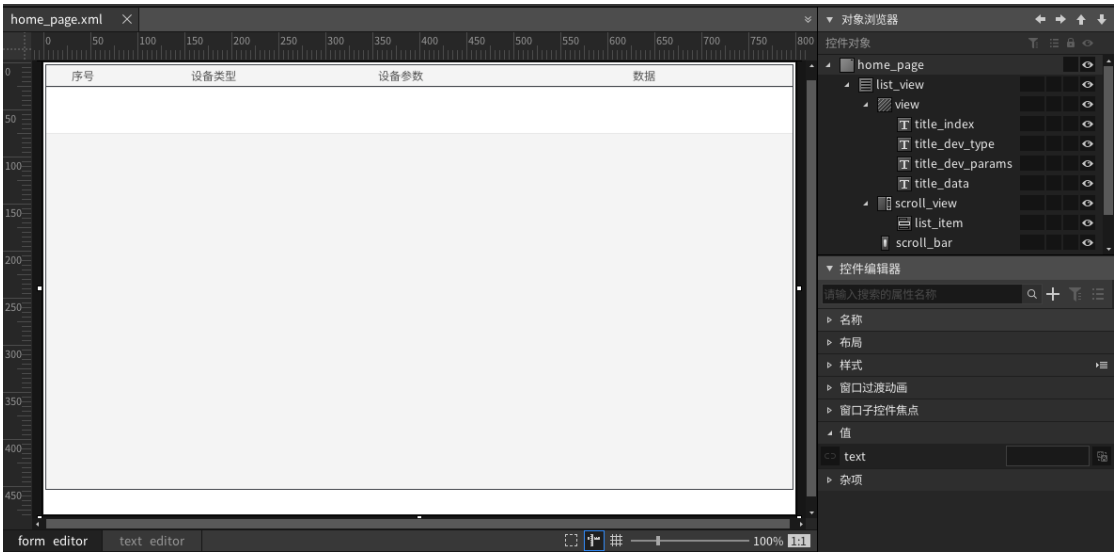

图 3.6 设备列表界面

3.3.1 列表渲染

列表渲染可将指定控件作为模板，将列表中的各项数据进行重复渲染。下一步将视图模型 home_page_view_model 中的 items 设备列表数组绑定到 list_item 列表项控件的列表渲染规则中：

步骤一：点击下图框选位置的按钮：

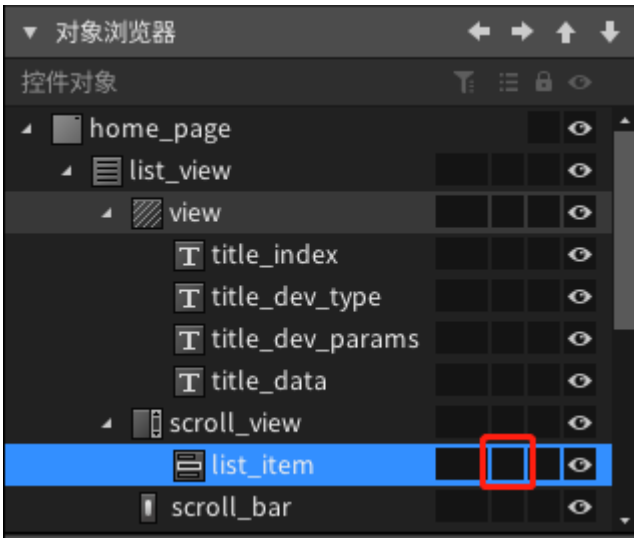

图 3.7　为列表项控件设置列表渲染规则步骤一

步骤二：在”绑定的数组”中点击选择 items 设备列表数组（上文在视图模型中添加的属性），或者点击右侧按钮进行更详细地设定，之后按确定完成绑定：

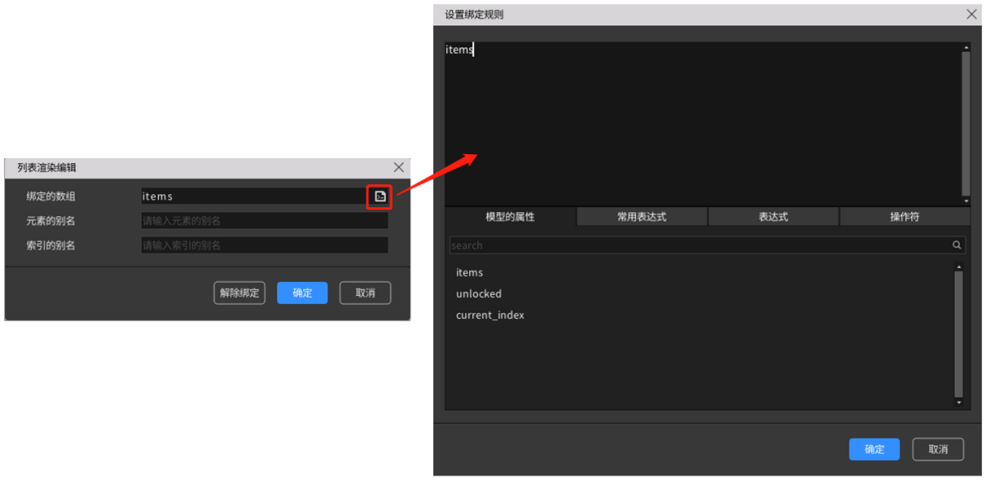

图 3.8　为列表项控件设置列表渲染规则步骤二

列表渲染设置完成后，在 list_item 控件对象中点击的下图红色处就会有个小图标显示：

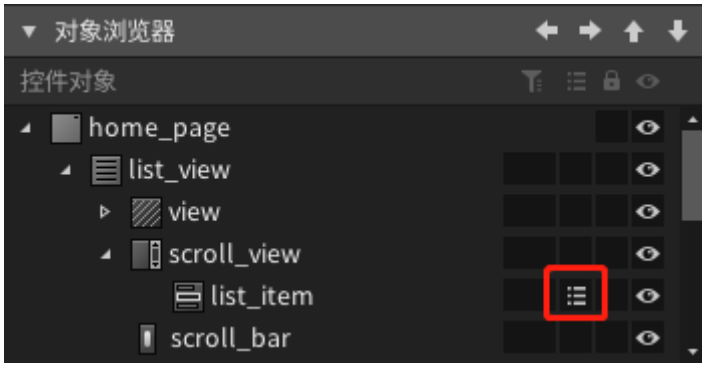

图 3.9　为列表项控件设置列表渲染规则完成

接下来在 list_item 控件及其子控件中可以通过 index 变量名访问当前项下标，item 变量名访问当前项的数组元素，在该案例项目里其元素就是 device 模型对象。

注意：list_item 是界面中的列表项控件；items 是在视图模型中添加的 object_array 类型属性，意为设备列表数组，用于存取设备；item 为条件渲染规则中用于获取绑定变量数组当前项元素的属性，注意不要混淆。

下一步给 list_item 设置子控件布局，在下文会有作用：

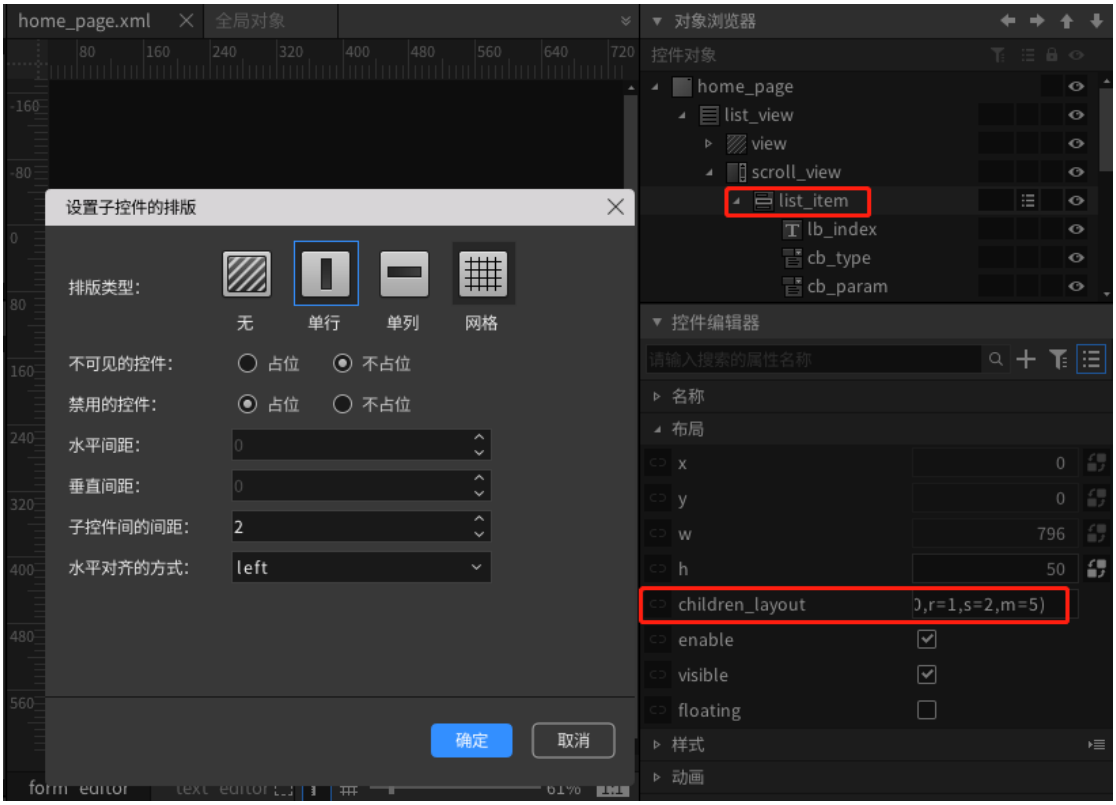

图 3.10　设置子控件布局

下一步给 list_item 增加子控件，通过数据绑定（步骤详见上文 2.2.3 章节），就可以在子控件显示列表当前项下标和设备信息了。

label 控件的 text 属性绑定 index 变量用于显示列表当前项下标：

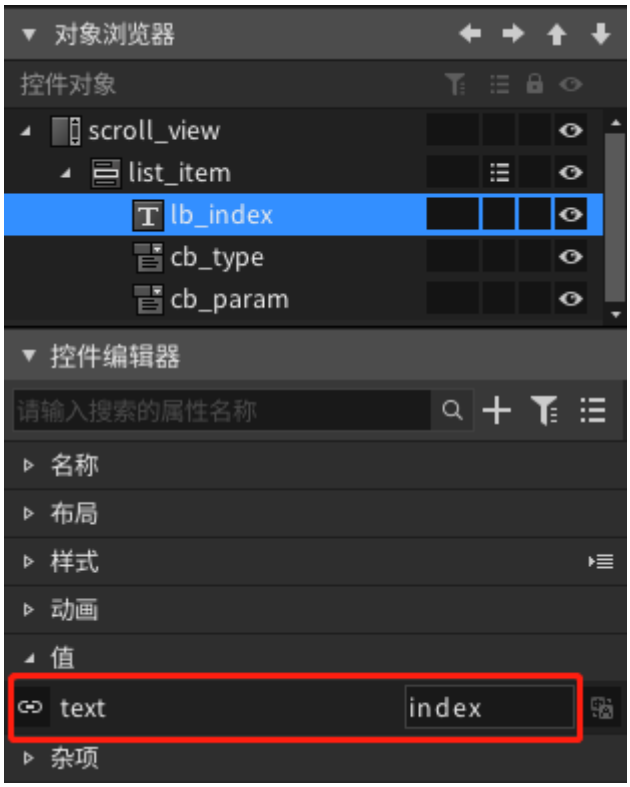

图 3.11　绑定当前项下标

combo box 控件的 value 属性绑定 item.pack_type 变量用于给用户选择当前项设备的设备类型（item.pack_type 表示 device 模型对象的 pack_type 属性）：

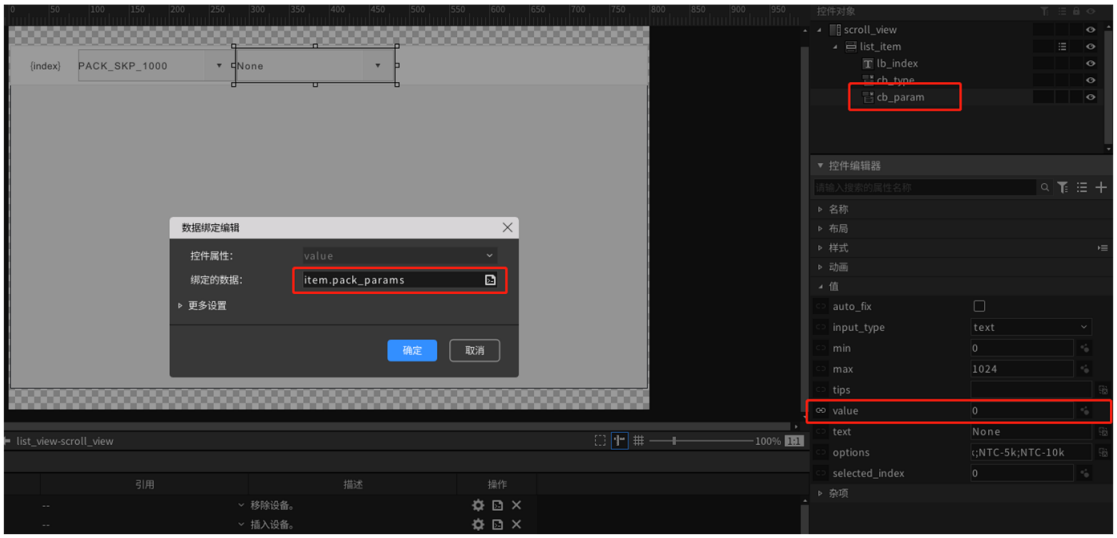

图 3.12　绑定设备模型属性

按照以上方法将设备固定部分的信息绑定绑定完成后，效果如下图：

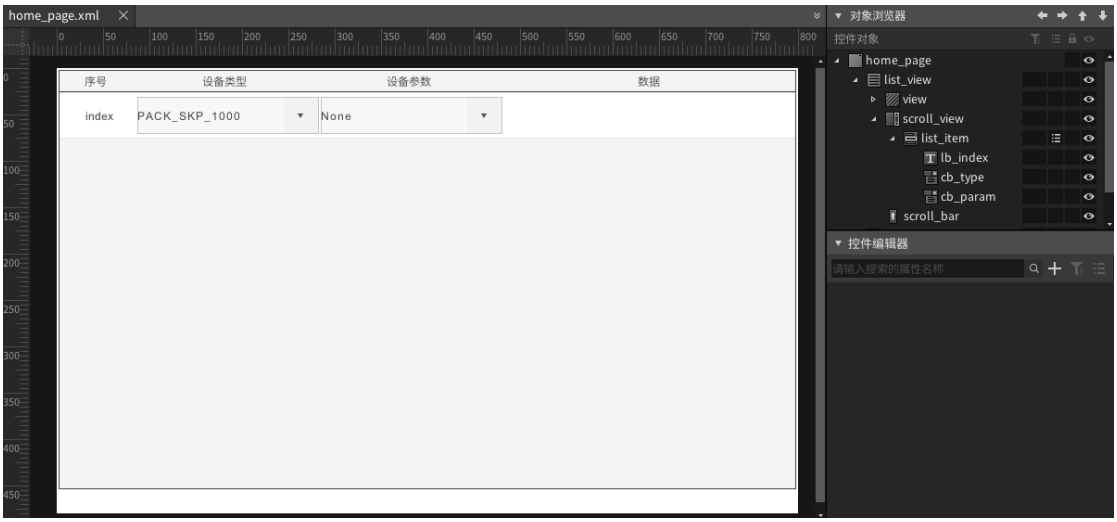

图 3.13　在列表项中显示设备固定部分的信息

还有其他设备数据会根据设备类型的不同，所显示的内容也不同，这时需要用到条件渲染。

3.3.2　条件渲染

条件渲染可以根据不同的条件进行不同的渲染。下一步使用条件渲染实现不同设备类型显示不同数据的功能点：

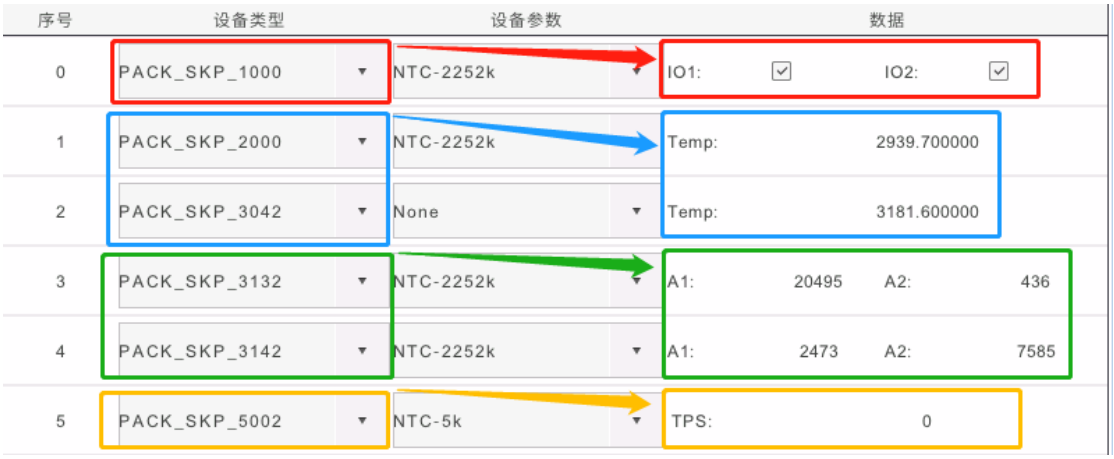

图 3.14　不同设备类型显示不同数据

要想实现上图的效果，首先设计用于显示设备数据的控件组合，并用容器 view 将其归类：

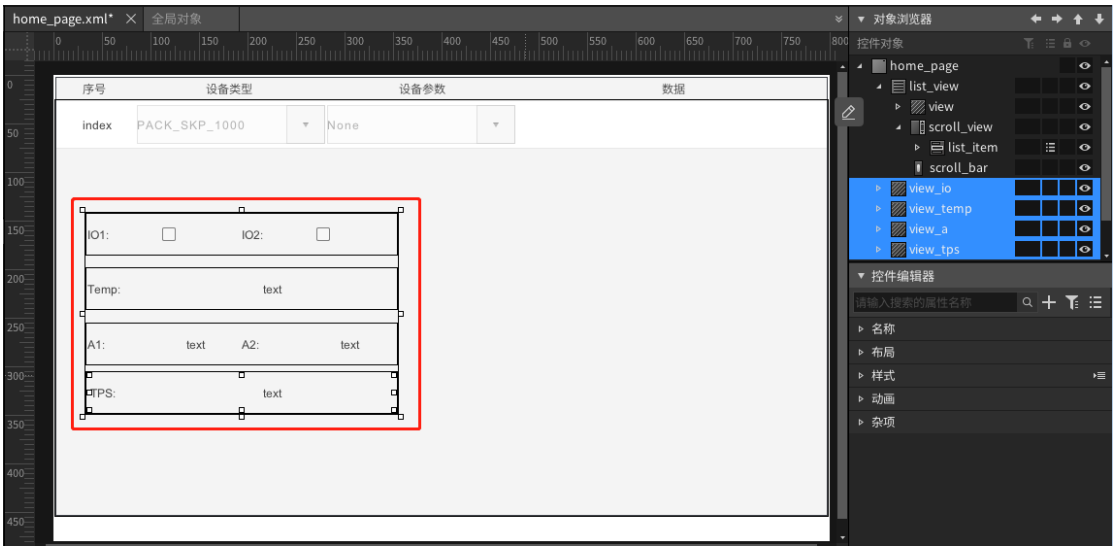

图 3.15　显示设备数据的控件组合

将这四个容器放到 list_item 中，在 list_item 子控件布局的作用下，会发现这四个容器会超出 list_item 的显示范围：

图 3.16　显示设备数据的控件组合

下一步给这四个容器设置条件渲染规则，运行时只有符合条件的那个容器才会被创建，从而达到不同条件下显示不同内容的效果。

步骤一：选择第一个容器，点击框选按钮：

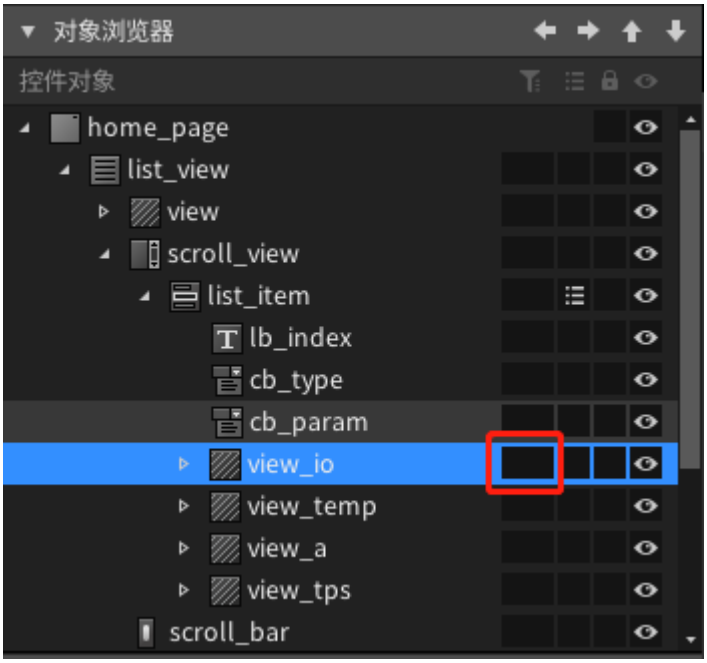

图 3.17　设置条件渲染步骤一

步骤二：在弹出对话框里点击框选处：

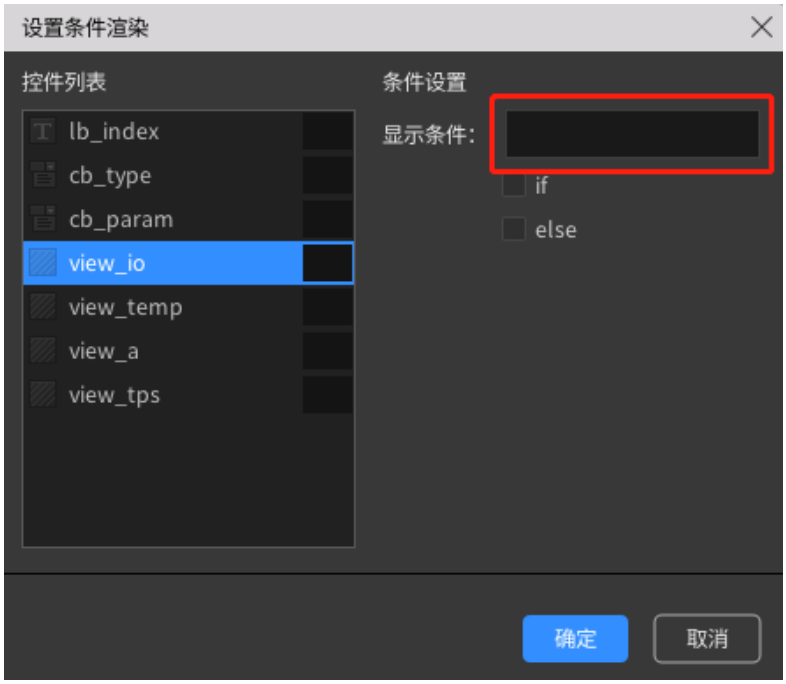

图 3.18　设置条件渲染步骤二

步骤三：在弹出对话框中设置渲染条件，图中框选条件意思为当前设备类型为 0（DEV_TYPE_PACK_SKP_1000），这渲染该控件，设置完成后点击确定：

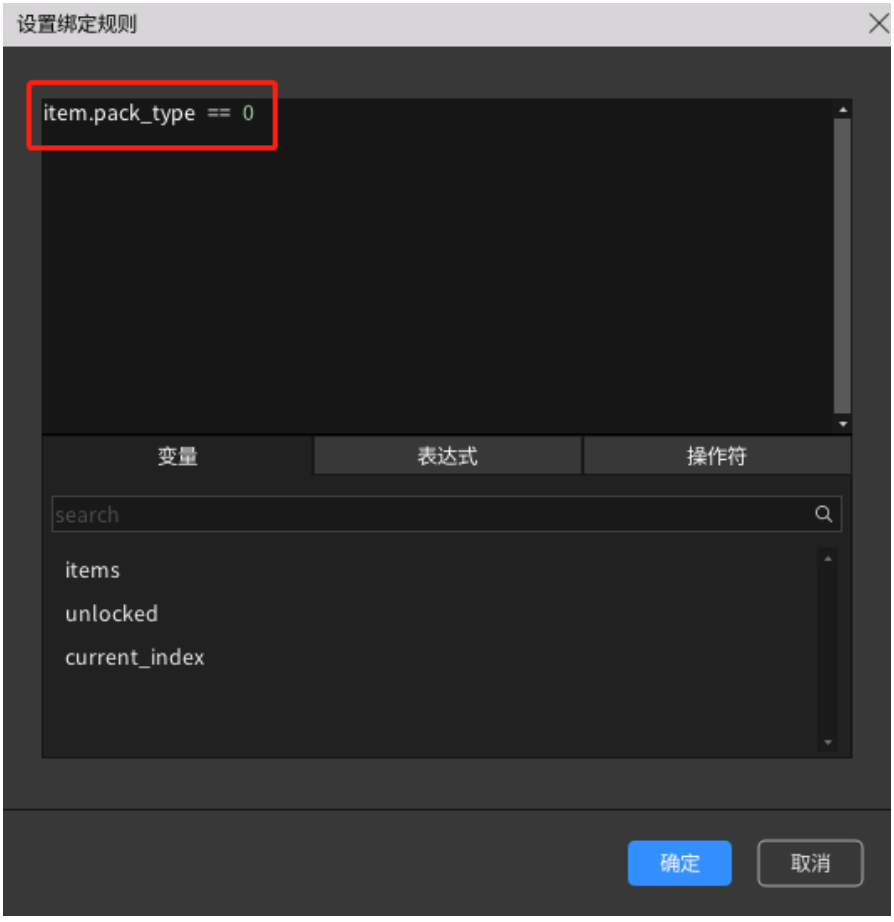

图 3.19　设置条件渲染步骤三

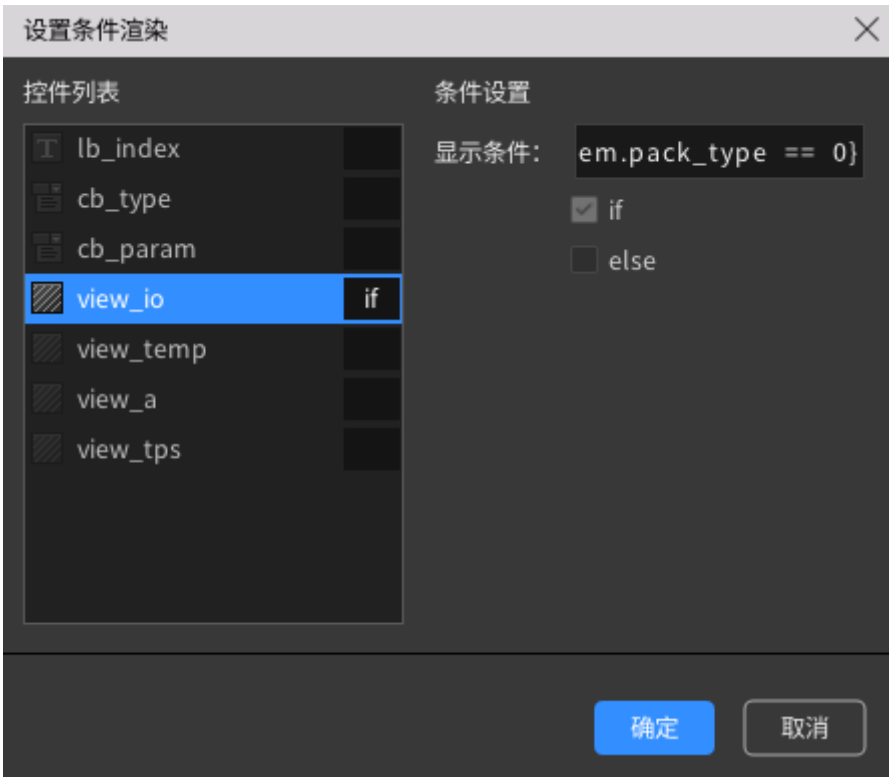

图 3.20　设置第一个容器条件渲染规则完成

步骤四：接着设置下一个容器，和步骤二、步骤三的操作相同：

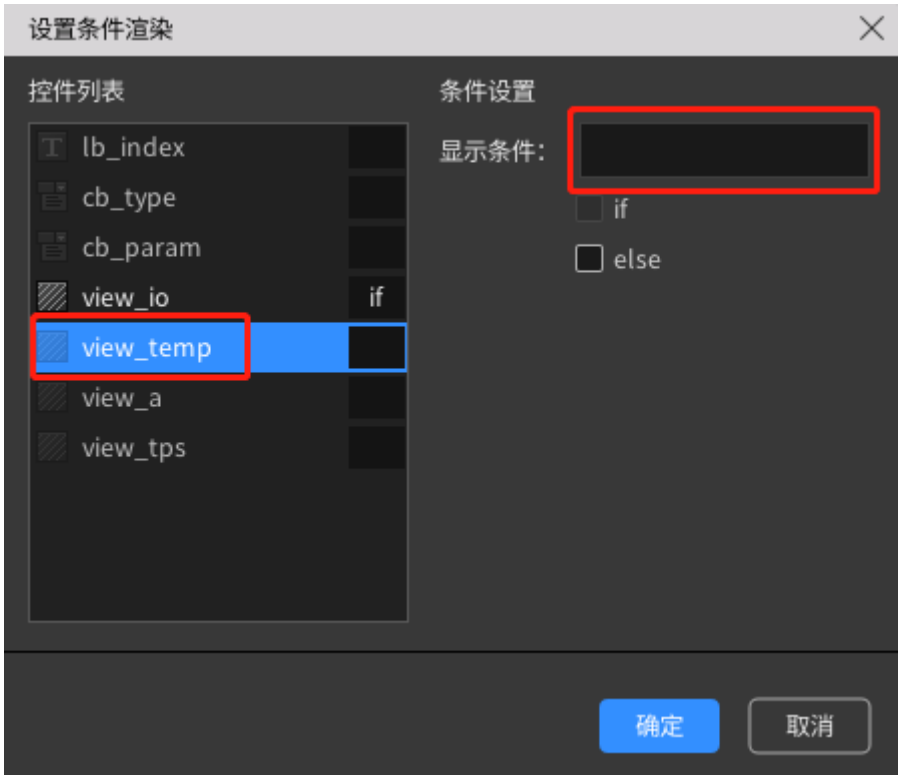

图 3.21　设置条件渲染步骤四

步骤五：到最后一个容器，就可以选择 else 判断，在前三个容器都不符合条件时，就会

渲染该容器，之后点击确定按钮保存渲染规则：

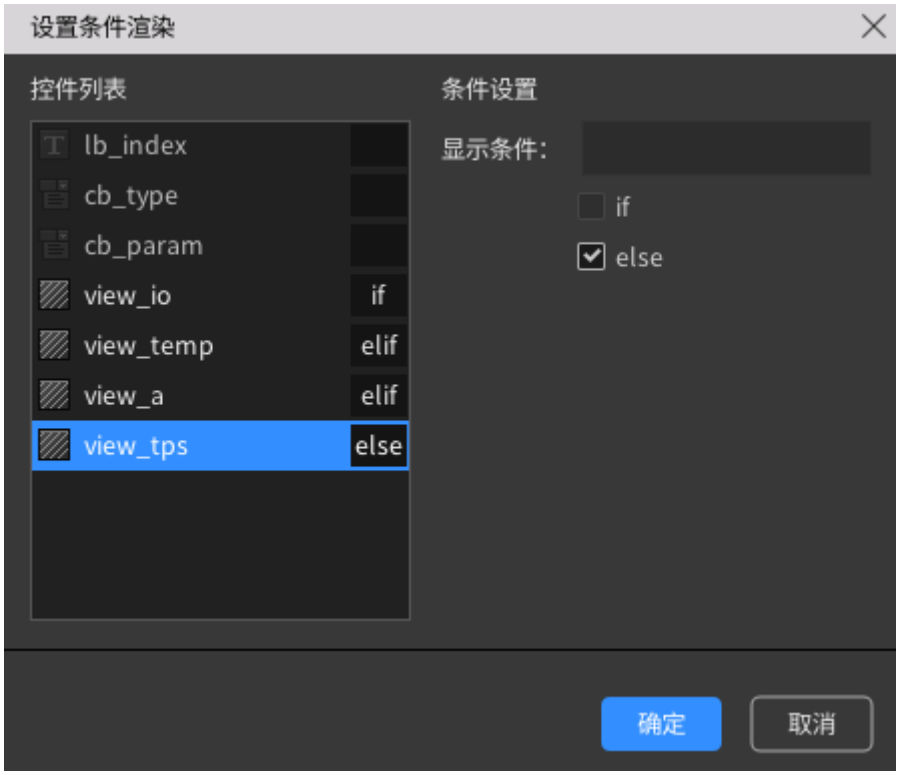

图 3.22　设置条件渲染步骤五

设置完成后，就会出现图中框选效果：

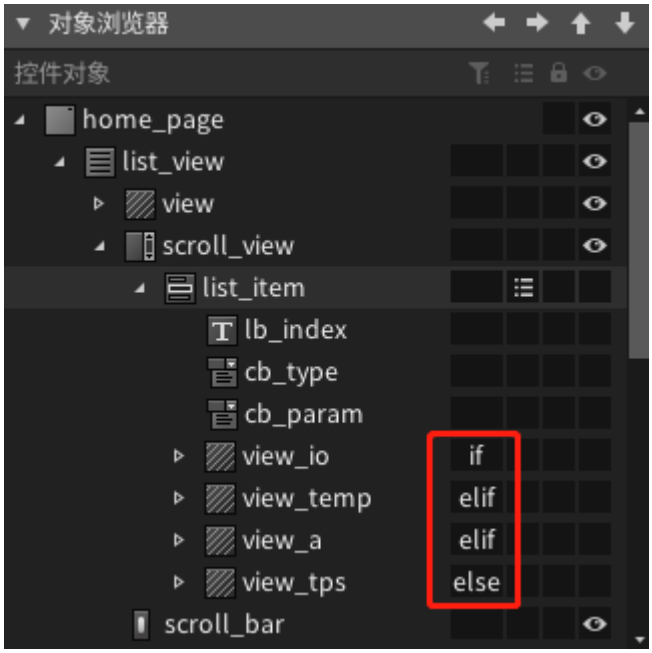

图 3.23　设置条件渲染完成

之后为容器中控件绑定当前项设备的设备属性即可，如将设备中的 io2 属性绑定到 check button 控件的 value 属性上：

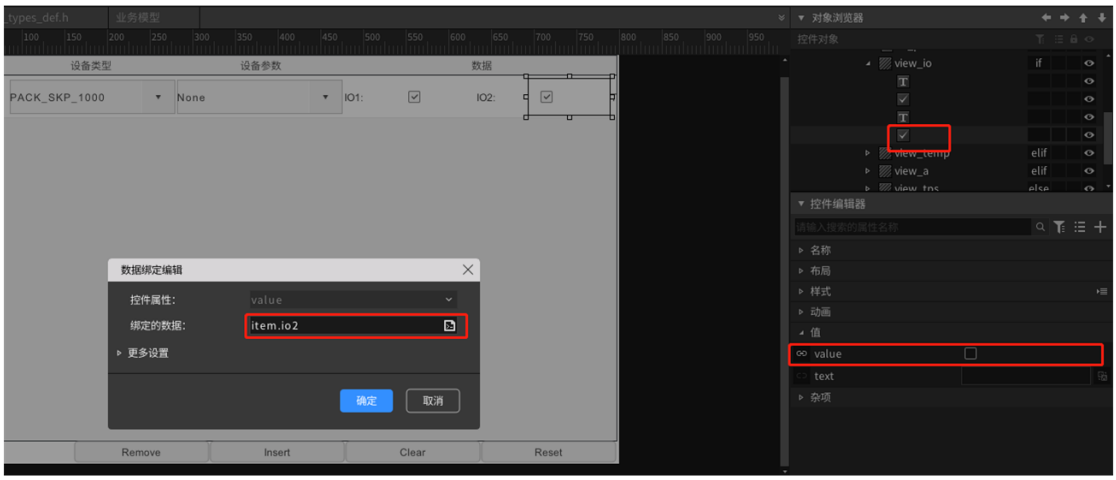

图 3.24　数据绑定

如将设备中的 temp 属性绑定到 label 控件的 text 属性上，并对显示内容格式化：

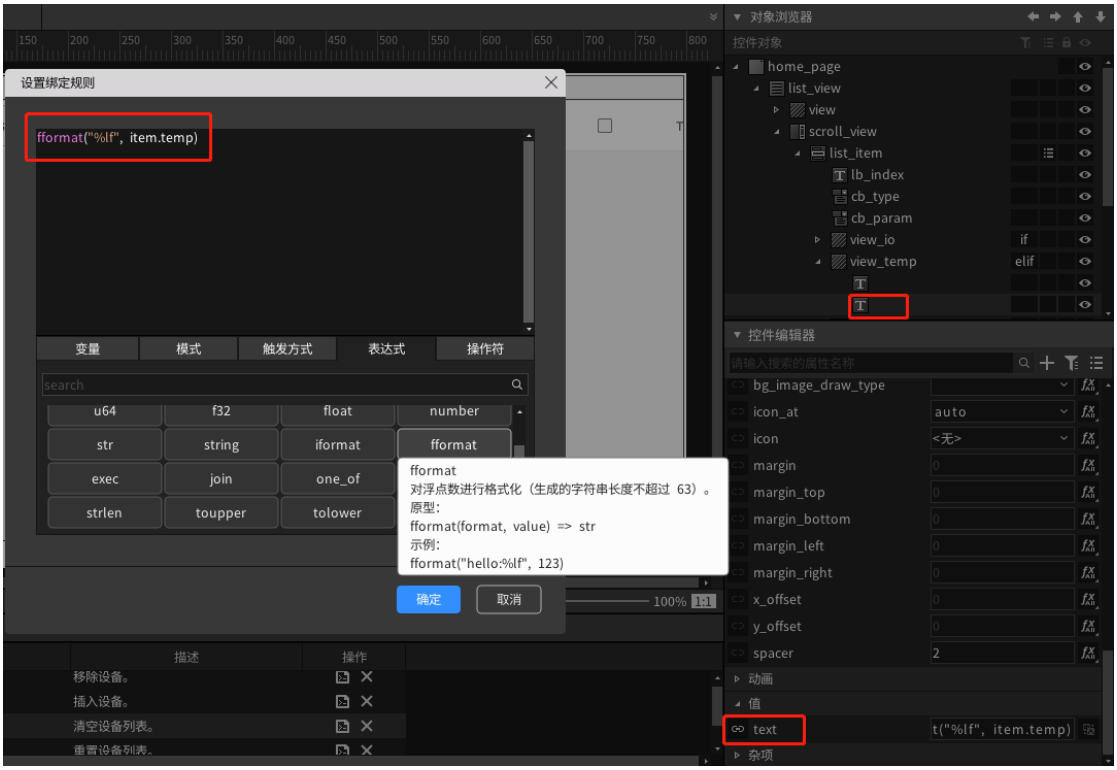

图 3.25　数据绑定

目前为止，已经实现了显示多个设备和不同设备类型显示不同数据的功能点，下一步实现解锁后才可以修改设备类型和设备参数这一功能：

增加 check button 控件，将 unlocked 属性绑定到 check button 控件的 value 属性：

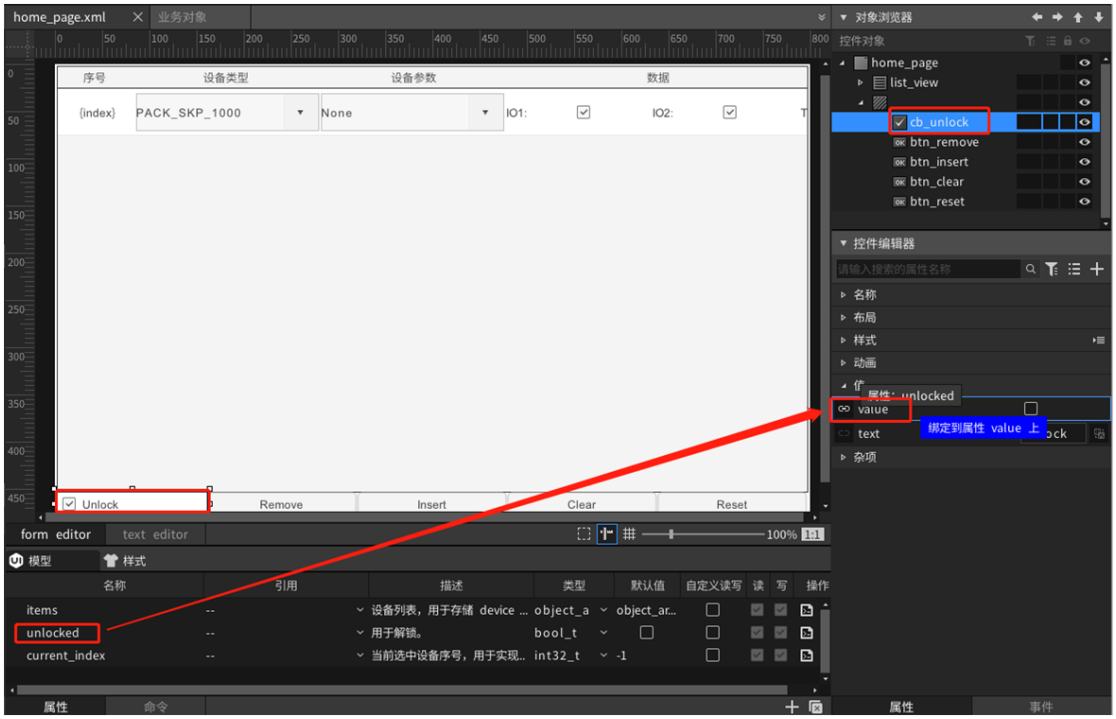

图 3.26　数据绑定

再将 unlocked 属性绑定到用于选择设备类型的 combo box 的 enable 属性，就可以实现通过 check button 控制设备类型能否被修改（控制设备参数能否被修改和该操作类似，这里就不赘述了）：

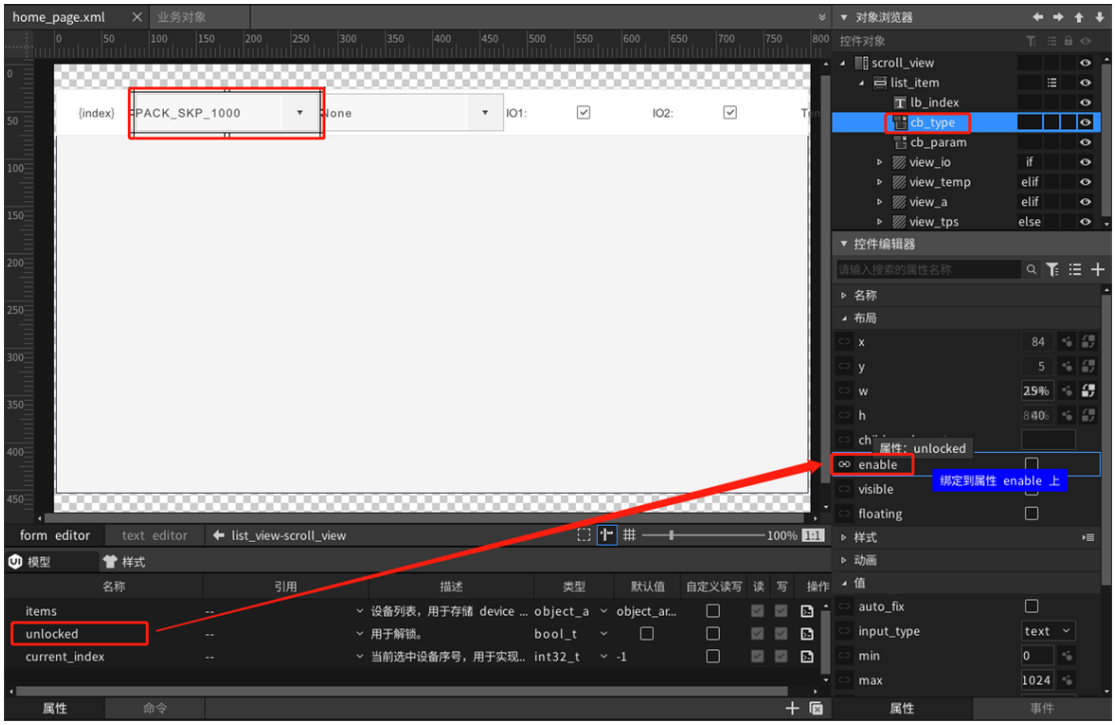

图 3.27　数据绑定

下一步实现设备的增加和删除功能，首先增加两个按钮，一个用于增加并插入设备，一个用于删除选中设备，绑定在视图模型设计好的 insert 命令和 remove 命令到对应的按钮的

点击事件上，以绑定 insert 命令为例：

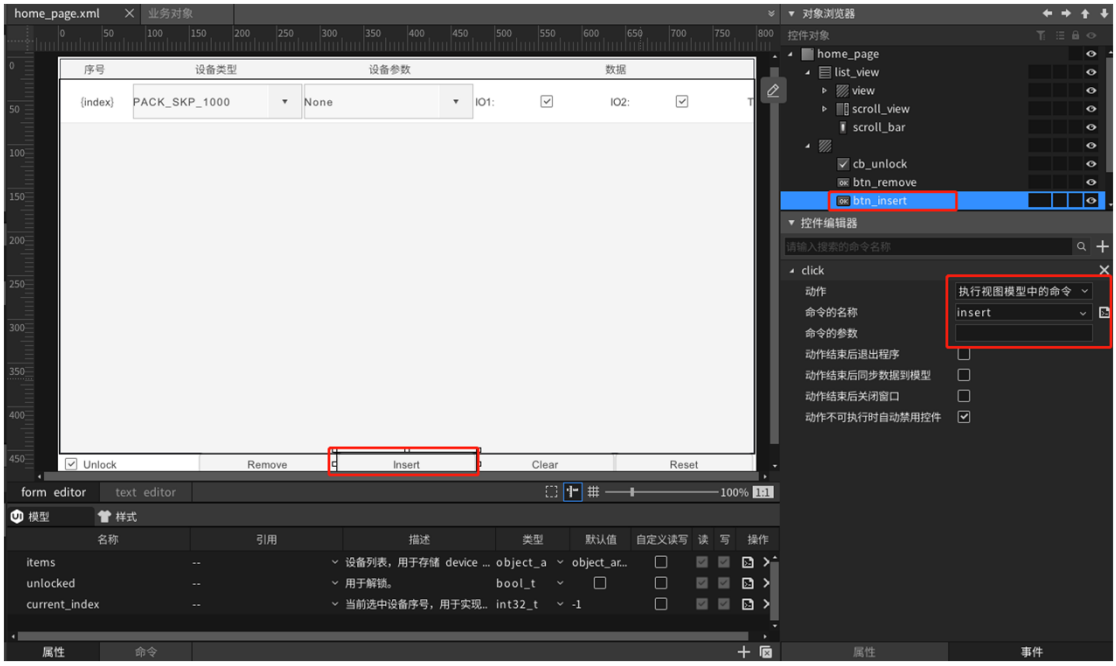

图 3.28　命令绑定

还需要将 setCurrent 命令绑定到 list_item 的点击事件上，实现通过点击 list_item 选中设备插入或删除序号（这里的参数是以 fscript 表达式形式的参数序列传入的，其意思是给名称为”index”参数赋值为列表当前项下标（index）, 想了解更多请参阅 AWTK-MVVM 命令绑定 [15]）：

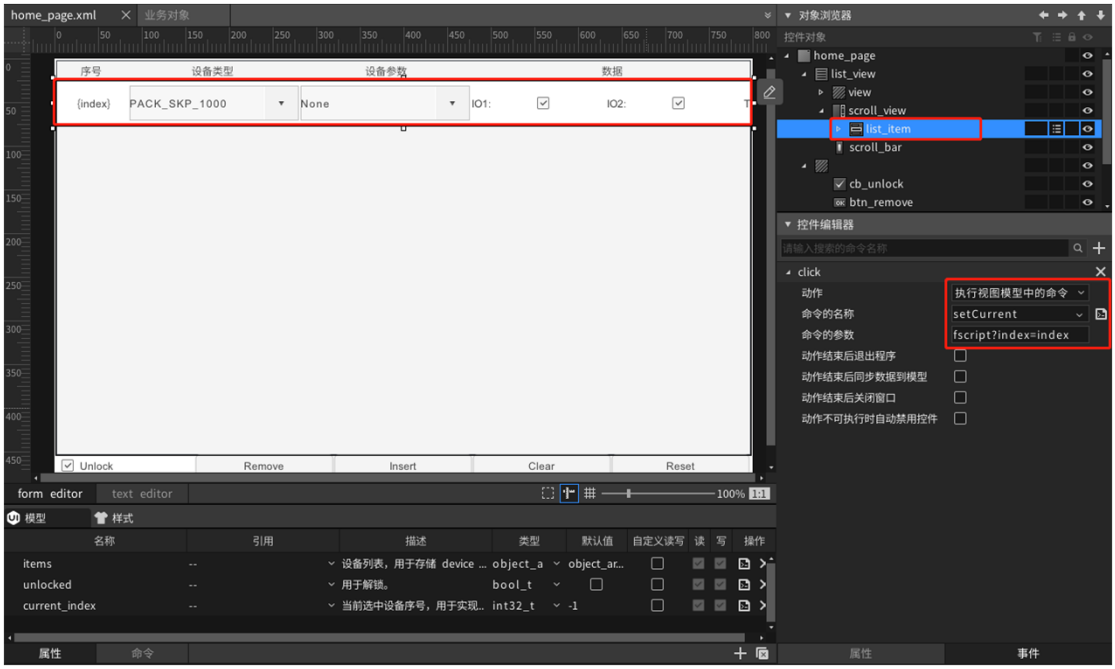

图 3.29　命令绑定

最后一步实现还原和清空设备列表功能：在界面处增加两个按钮，一个用于还原设备列

[15] https://github.com/zlgopen/awtk-mvvm/blob/master/docs/11.command_binding.md

表，一个用于清空设备列表，绑定在视图模型设计好的 reset 命令和 clear 命令到对应的按钮的点击事件上即可，以绑定 clear 为例：

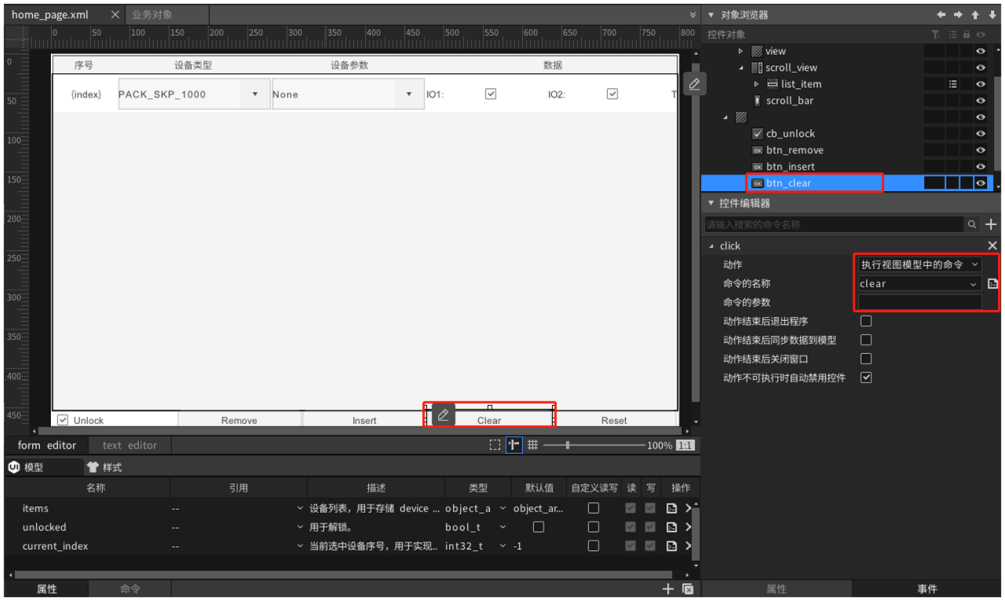

图 3.30　命令绑定

到这一步，案例项目的所有功能就完成了。

3.4　运行效果

最后将项目打包并编译，点击模拟运行，即可看到其运行效果：

图 3.31　操作步骤

图 3.32　运行案例项目

4. 制作 MVVM-JS 项目

前面我们介绍了如何制作 MVVM-C 项目，由于 C 语言是静态语言，没有办法直接通过名字去访问对象的成员变量和成员函数，所以 ViewModel 的代码会比较繁琐。如果用 JS 语言开发 MVVM 项目则代码会更加简洁。

顾名思义，**MVVM-JS** 项目是使用 **AWTK-MVVM** 框架和 JS 语言开发应用程序的一个工程项目。更多关于 MVVM-JS 的介绍，请参阅 awtk-mvvm/docs/13.js_model[16]。下文将详细介绍如何使用 AWTK Designer 制作一个 MVVM-JS 项目。

由于在 AWTK Designer 中制作 MVVM-JS 项目的操作与 MVVM-C 项目类似，因此这里仅简单介绍新建与编辑 MVVM-JS 的操作步骤，具体操作细节详见本文第二章，下文主要对比二者在代码上的区别。

注：目前 MVVM-JS 还无法实现一些平台相关的功能，比如访问硬件、网络、文件系统和异步化功能，后续随着 awtk-iotjs[17] 的完善，MVVM-JS 项目将支持导入 iotjs 的相关模块实现这些功能。

4.1 新建 MVVM-JS 项目工程

新建 MVVM-JS 项目的操作与 MVVM-C 类似，点击 AWTK Designer 中的”新建项目”按钮，打开”新建项目”对话框，然后选择”MVVM-JS”项目类型，设置好相关参数后，点击”创建”按钮即可，如下图所示：

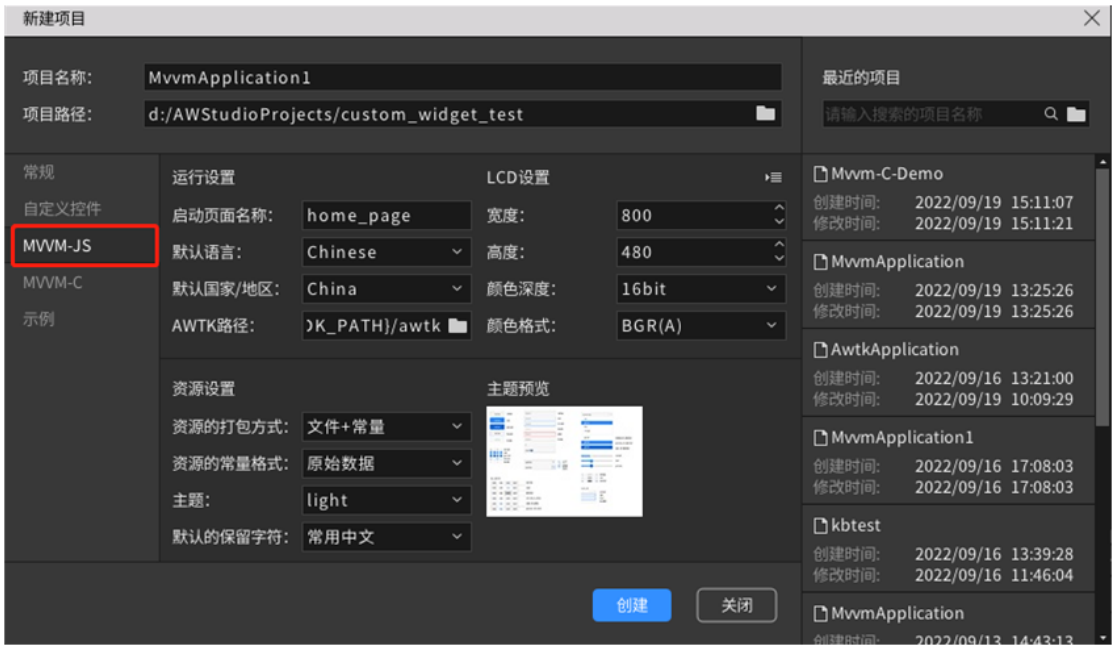

图 4.1 新建 MVVM-JS 项目

新建项目后，AWTK Designer 同样会进入主界面，默认创建并打开 home_page 页面（View），而且项目的目录结构与也与 MVVM-C 项目的类似，只不过 models 和 view_models 目录中改为存放 JS 代码并且新增了 src/application.js 文件，用于处理程序启动和退出的过程，

[16] https://github.com/zlgopen/awtk-mvvm/blob/master/docs/13.js_model.md
[17] https://github.com/zlgopen/awtk-iotjs/

代码如下：

```
/* src/application.js */
/* 导入全局对象管理器模块 */
import {globalObjectsManager} from "./models/objects_manager";

/**
 * @class Application
 * 注册一个Application实例，用于指定程序的生命周期函数。
 * Application()方法全局只能调用一次。
 */
Application({
  /**
   * @method onLaunch
   * 当程序初始化完成时调用，全局只触发一次。
   */
  onLaunch : function() {
    console.log('============== launch ==============');
    /* 创建全局对象管理器 */
    if (typeof globalObjectsManager.onCreate === 'function') {
      globalObjectsManager.onCreate();
    }

    /* 此处可以编写程序启动时的 init 代码 */
    ......
  },
  /**
   * @method onExit
   * 当程序退出时调用，全局只触发一次。
   */
  onExit : function() {
    /* 此处可以编写程序退出时的 deinit 代码 */
    ......

    if (typeof globalObjectsManager.onDestroy === 'function') {
      globalObjectsManager.onDestroy();
    }
    console.log('==============  exit  ==============');
  }
});
```

4.2 ViewModel（视图模型）

ViewModel（视图模型，简称 VM）的介绍详见上文 2.2 章节，MVVM-JS 项目中创建 VM 以及在 AWTK Designer 中对 VM 进行的各种操作都与 MVVM-C 项目类似，二者的区别仅在于 VM 的实现语言不同。

4.2.1 为 View 新增 ViewModel

MVVM-JS 项目创建 VM 的步骤与 MVVM-C 项目类似，详见上文第二章，此处为 home_page 页面新增一个 VM，如下图所示：

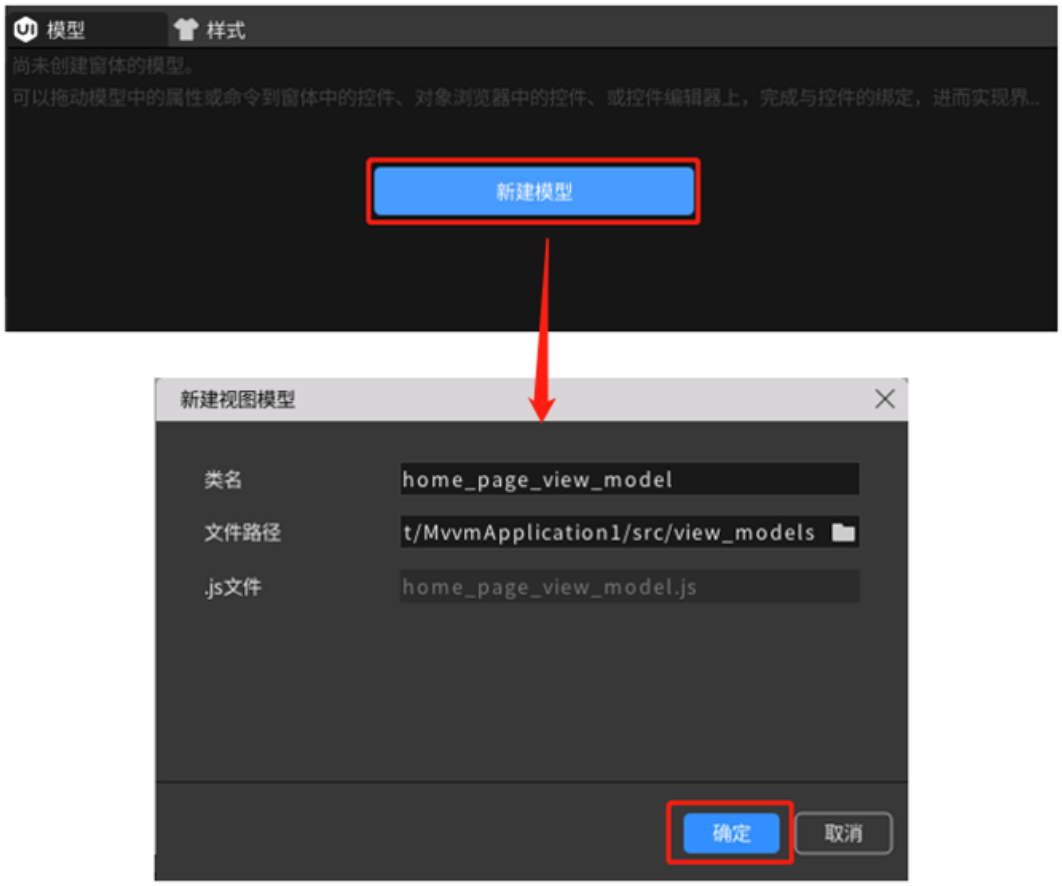

图 4.2 新建 ViewModel

每个页面都可以创建不同的 VM，创建完成后会在项目 src/view_models 目录下新增对应的 JS 代码文件，此处为 home_page_view_model.js 文件，代码如下，可以看出 JS 实现的 ViewModel 要简洁很多：

```
/* src/view_models/home_page_view_model.js */
/* 导入全局对象管理器模块 */
import {globalObjectsManager} from "../models/objects_manager";

/**
 * @class home_page_view_model
 * @parent ViewModel
```

```
 * @annotation ["model", "view_model", "custom_prop"]
 */
ViewModel('home_page_view_model', {
  data: {  /* ViewModel 中的属性 */
  },
  computed: {
    /**
     * @property {ObjectsManager} _global
     * @annotation ["readable", "defvalue:globalObjectsManager", "from_global"]
     * 全局对象管理器，提供访问业务数据和操作的接口，一个程序有且仅有一个，
     * 被ViewModel共享，默认在程序启动时创建、退出时销毁。
     */
    _global: {
      get: function() {
        return globalObjectsManager;
      }
    }
  },
  methods: {  /* ViewModel 中的命令 */
  },
  onCreate: function(req) {    /* ViewModel 创建时的回调函数 */
    return RET_OK;
  },
  onDestroy: function() {      /* ViewModel 销毁时的回调函数 */
    return RET_OK;
  },
  onWillMount: function(req) { /* ViewModel 即将挂载时的回调函数 */
    return RET_OK;
  },
  onMount: function() {        /* ViewModel 挂载时的回调函数 */
    return RET_OK;
  },
  onWillUnmount: function() {  /* ViewModel 即将卸载时的回调函数 */
    return RET_OK;
  },
  onUnmount: function() {      /* ViewModel 卸载时的回调函数 */
    return RET_OK;
  }
});
```

4.2.2 为 ViewModel 添加属性

在 MVVM-JS 项目中，同样也可以为 ViewModel 添加局部属性和全局属性，它们的添加步骤与 MVVM-C 项目一样，详见上文第二章。

此处用同样的步骤为 home_page_view_model 创建 Number 类型的局部属性 property 和全局属性 g_prop，并将它们分别初始化为 40 和 60，如下图所示：

图 4.3 为 ViewModel 添加属性

添加成功后可以分别在 home_page_view_model.js 和 objects_manager.js 中看见对应的局部属性和全局属性，代码如下：

```
/* src/view_models/home_page_view_model.js */
ViewModel('home_page_view_model', {
  data: {
    /**
     * @property {Number} property
     * @annotation ["readable", "writable", "defvalue:40"]
     */
    property: 40
  },
  ......
});
```

```
/* src/models/objects_manager.js */
export class ObjectsManager {
  constructor() {
    /**
     * @property {Number} g_prop
     * @annotation ["readable", "writable", "defvalue:60"]
     */
    this.g_prop = 60;
  }
  ......
}
```

4.2.3 数据绑定

MVVM-JS 项目中的数据绑定与 MVVM-C 项目一样，此处同样将上文添加的 property 属性绑定到 slider 控件的 value 属性上为例，简单演示两种绑定方式。

方式一：在控件编辑器中实现数据绑定。

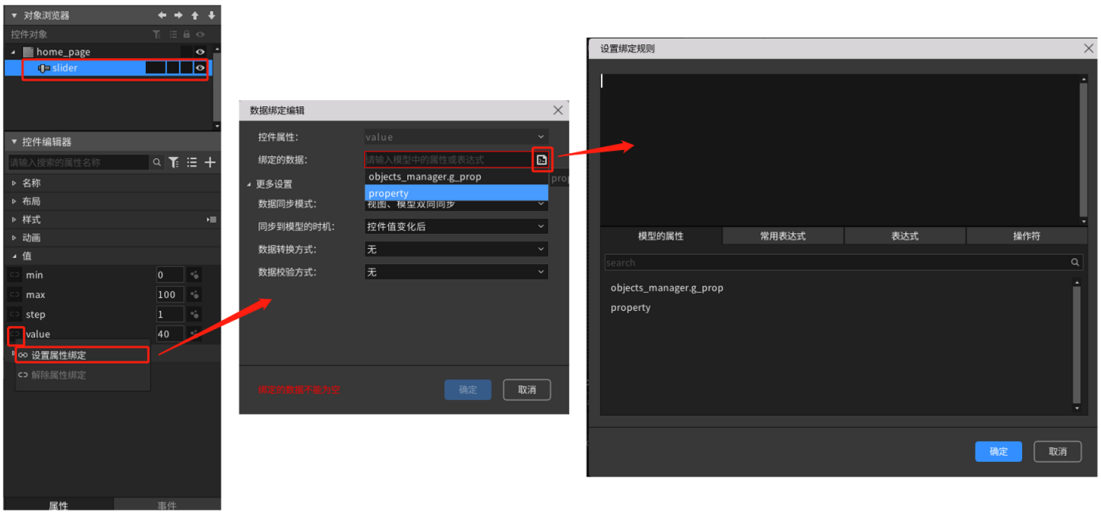

图 4.4 在控件编辑器中实现数据绑定

方式二：通过拖拽模型编辑器中的属性实现数据绑定。

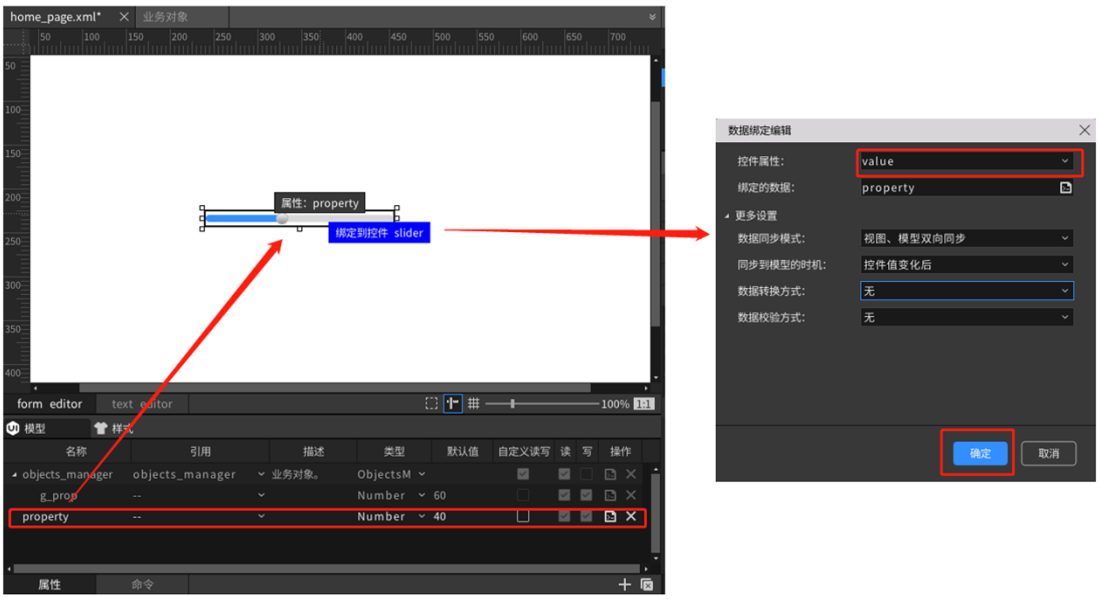

图 4.5 通过拖拽模型编辑器中的属性实现数据绑定

以上两种方式任选一种，绑定成功后，再创建一个 label 控件，用同样的方法将这个 label 控件的 text 属性与 property 属性绑定起来。绑定完成后 home_page.xml 的代码如下：

```
<window v-model="home_page_view_model" name="home_page">
  <slider name="slider" x="c" y="m" w="214" h="16" v-data:value="{property}"/>
  <label name="label" x="c" y="200" w="160" h="28" v-data:text="{property}"/>
```

```
</window>
```

点击 AWTK Designer 中的”模拟运行”，当 slider 控件的 value 改变时，label 控件的显示文本也会跟着一起变化，如下图所示：

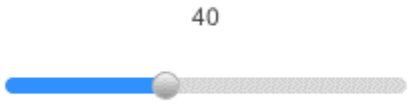

图 4.6　数据绑定成功

4.2.4　为 ViewModel 添加命令

在 MVVM-JS 项目中，为 ViewModel 添加命令的步骤与 MVVM-C 项目一样，也可以添加局部命令和全局命名，具体详见上文第二章。

此处用同样的步骤为 home_page_view_model 添加局部命令 command 和全局命令 g_command，如下图所示：

图 4.7　为 ViewModel 添加命令

添加成功后可以分别在 home_page_view_model.js 和 objects_manager.js 中看见对应的局部命令和全局命令，此处让它们分别打印不同的信息，代码如下：

```
/* src/view_models/home_page_view_model.js */
ViewModel('home_page_view_model', {
  ......
  methods: {

    /**
     * @method command
     * @annotation ["command"]
     * 局部命令
     * @param {String|Object} args 命令的参数。
     * @return {TRET} 返回RET_OK表示成功，否则表示失败。
     */
    canCommand: function(args) {
      return true;
    },
```

```
    command: function(args) {
      console.log('command run!\n'); /* 打印信息 */
      /* 其他处理代码 */
      ......
      return RET_OK;
    }
  },
  ......
});
```

```
/* src/models/objects_manager.js */
export class ObjectsManager {
  ......
  /**
   * @method g_command
   * @annotation ["command"]
   * 全局命令
   * @param {String|Object} args 命令的参数。
   * @return {TRET} 返回RET_OK表示成功，否则表示失败。
   */
  canG_command(args) {
    return true;
  }

  g_command(args) {
    console.log('g_command run!\n'); /* 打印信息 */
    /* 其他处理代码 */
    ......
    return RET_OK;
  }
}

/* 导出全局对象管理器模块 */
export var globalObjectsManager = new ObjectsManager();
```

4.2.5 命令绑定

MVVM-JS 项目的命令绑定步骤与 MVVM-C 项目一样，此处用同样的步骤将上文添加的 command 命令绑定到 silder 控件的”value_changed”事件上，如下图所示：

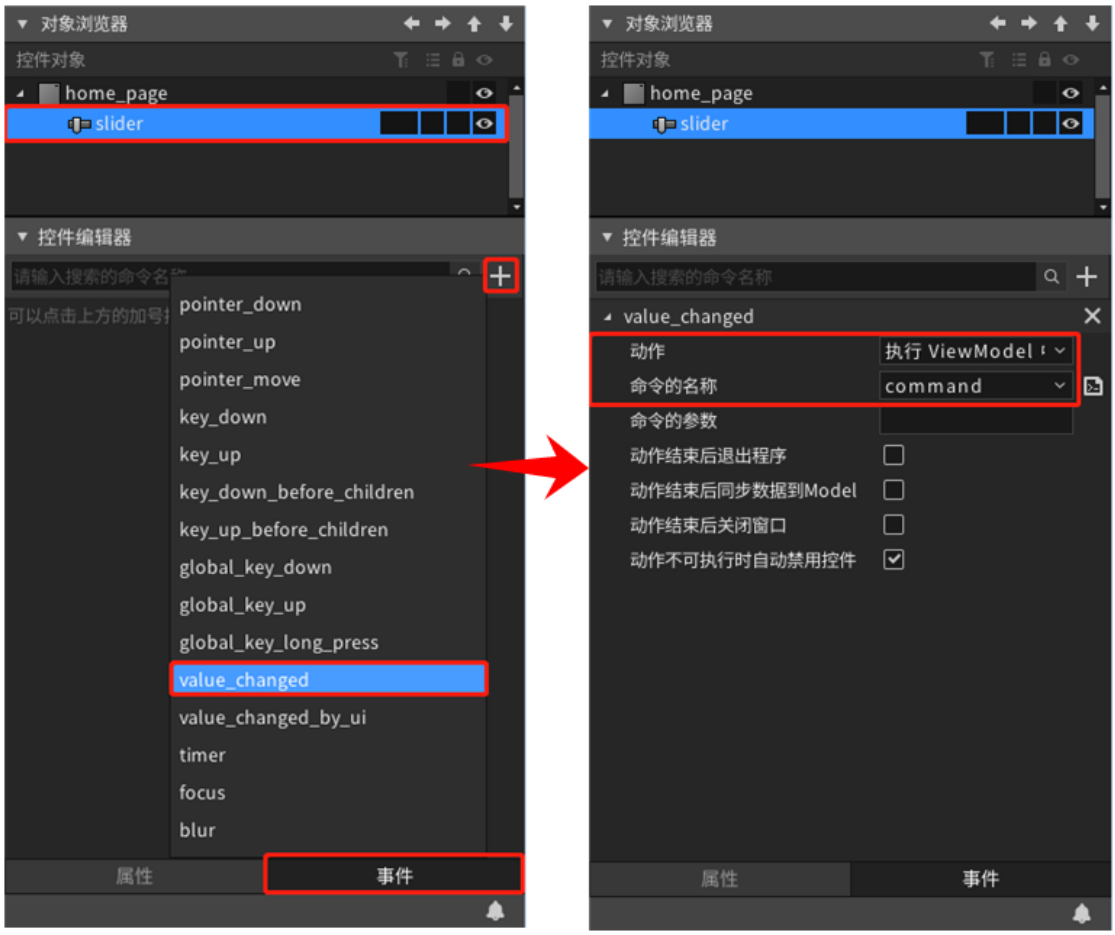

图 4.8　实现命令绑定

绑定完成后，点击 AWTK Designer 中的”模拟运行”，当 silder 控件的值发生变化后，将触发该事件并执行绑定的 command 命令打印相关信息，如下图所示：

图 4.9　执行 command 命令

关于命令参数的相关介绍详见上文 2.2.5 章节，在 MVVM-C 项目中，命令参数会以字符串的形式传递到代码中，需要进一步转化为 Object 对象，但在 MVVM-JS 项目中则不需要，args 可以是 String 类型和 Object 类型，代码如下，此处也可以看出 JS 语言在实现上更为简单明了：

```
/* src/view_models/home_page_view_model.js */
ViewModel('home_page_view_model', {
  methods: {
    /* 命令参数 args 类型为 String 或 Object */
    command: function(args) {
      console.log('command run!\n'); /* 打印信息 */
      return RET_OK;
    }
  },
  ......
});
```

4.3 Model（模型）

Model（模型，简称 M）的介绍详见上文 2.3 章节，MVVM-JS 项目中创建 M 以及在 AWTK Designer 中对 M 进行的各种操作都与 MVVM-C 项目类似，二者的区别仅在于 M 的实现语言不同

4.3.1 新建 Model

MVVM-JS 项目创建 M 的步骤与 MVVM-C 项目类似，详见上文第二章，此处用相同的步骤创建一个 model，如下图所示：

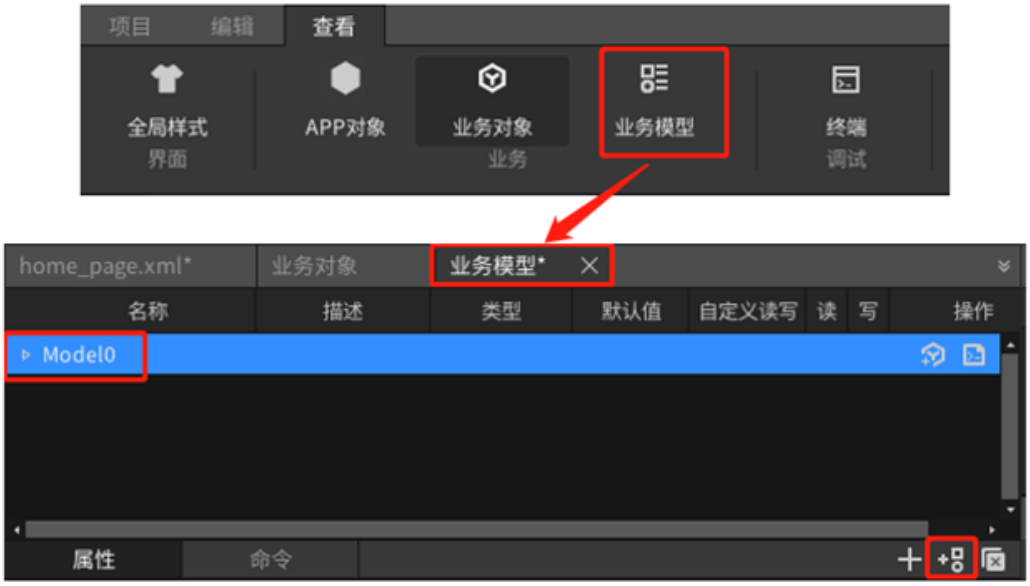

图 4.10 新建 Model

创建完成后会在项目 src/models 目录下新增对应的 JS 代码文件，此处为 model.js 文件，代码如下：

```
/* src/models/model.js */
/**
 * @class model
 * @parent Object
```

```
 * @annotation ["model", "custom_prop"]
 */
export class model {
  constructor() {
  }
}
```

4.3.2 为 Model 添加属性

在 MVVM-JS 项目中，为 Model 添加属性的步骤与 MVVM-C 项目一样，详见上文第二章，此处同样为 model 添加一个 Number 类型的 m_prop 属性，并将其默认值改为 50，如下图所示：

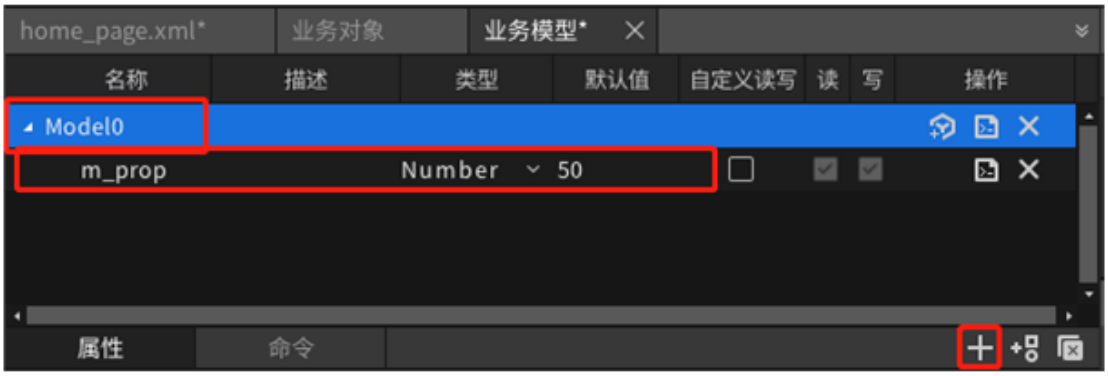

图 4.11 新建 Model

添加完成后，可以在 model.js 文件中看见对应的 m_prop 属性，代码如下：

```
/* src/models/model.js */
export class model {
  constructor() {
    /**
     * @property {Number} m_prop
     * @annotation ["readable", "writable", "defvalue:50"]
     */
    this.m_prop = 50;
  }
}
```

4.3.3 为 Model 添加命令

在 MVVM-JS 项目中，为 Model 添加命令的步骤与 MVVM-C 项目一样，详见上文第二章，此处同样为 model 添加一个 m_command 命令，如下图所示：

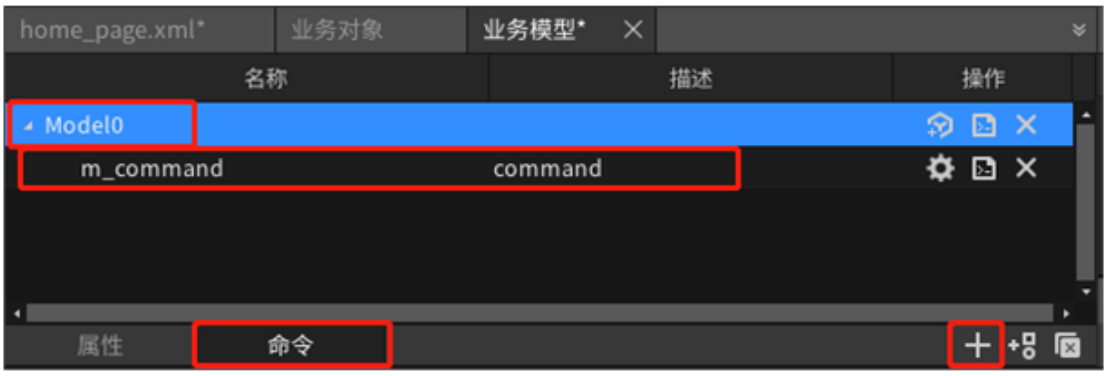

图 4.12　为 Model 添加命令

添加完成后，可以在 model.js 文件中看见对应的 m_command 命令，代码如下：

```
/* src/models/model.js */
export class model {
  ......
  /**
   * @method m_command
   * @annotation ["command"]
   * command
   * @param {String|Object} args 命令的参数。
   * @return {TRET} 返回RET_OK表示成功，否则表示失败。
   */
  canM_command(args) {
    return true;
  }
  m_command(args) {
    return RET_OK;
  }
}
```

4.3.4　添加 Model 对象到 ViewModel

添加 Model 对象到 ViewModel 实际上就是为 ViewModel 添加一个属性，属性类型选择创建的 Model 名称即可（例如上文的 model），此处将上文创建的 model 添加到 home_page_view_model 中，如下图所示：

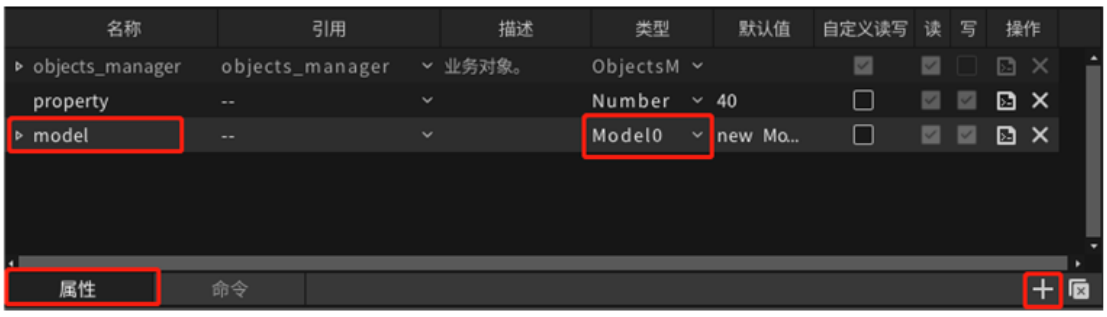

图 4.13　添加 Model 对象到 ViewModel

添加完成后即可在 home_page_view_model 中看见 model 的 m_prop 属性和 m_command

命令，如下图所示，它们的绑定步骤和 ViewModel 本身的属性没有区别，此处不多赘述。

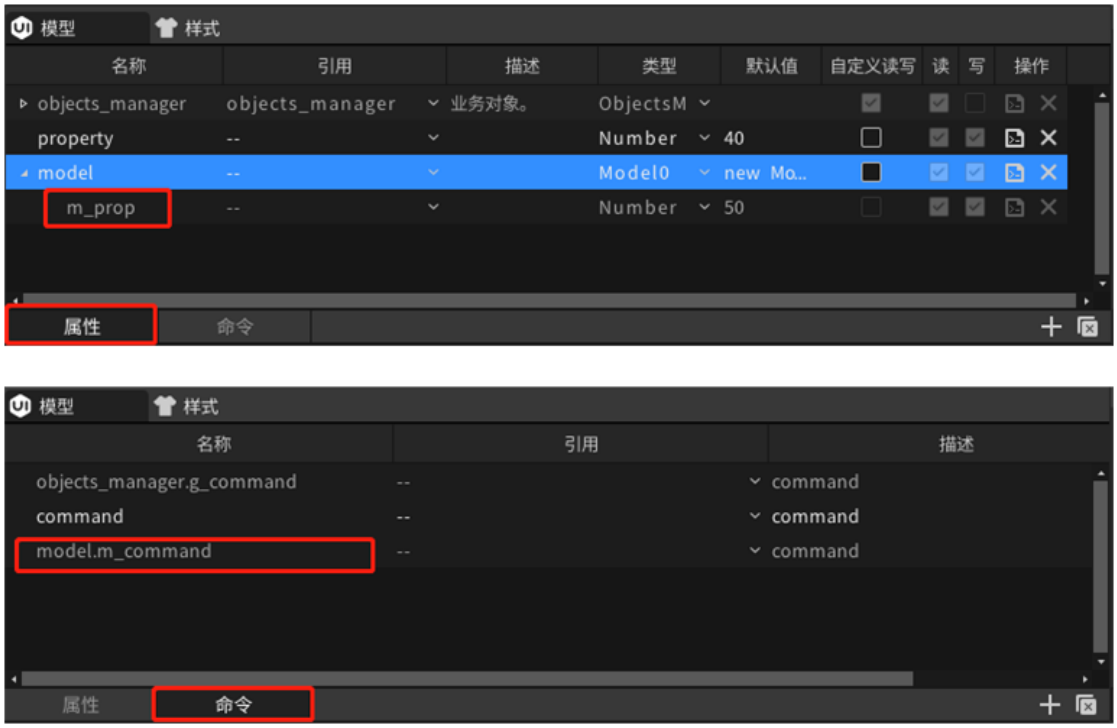

图 4.14　m_prop 属性和 m_command 命令

5. MVVM-JS 案例分析

上一章介绍了如何制作 MVVM-JS 项目，接下来本章将以 Mvvm-JS-Demo 为例，详细介绍使用 AWTK Designer 开发 MVVM-JS 项目过程，加深大家对 MVVM-JS 的理解，该案例的界面和功能跟第三章中介绍的 Mvvm-C-Demo 一样，此处不过多介绍，具体详见上文第三章，下文中将详细介绍二者在实现语言上的区别。

5.1 模型设计

首先使用 AWTK Designer 新建一个 MVVM-JS 项目工程，步骤详见上文 4.1 章节。根据 Mvvm-JS-Demo 的功能需求，此处我们需要添加一个名为”device”的设备模型，该模型与 3.1 章节中的介绍类似，包含设备类型、设备参数和数据，但由于底层采用 JS 语言实现，因此数据类型与 Mvvm-C-Demo 中的略微不同，详见下表：

属性名称	类型	描述	备注
pack_type	Number	设备类型	由于 MVVM-JS 项目暂不支持自定义枚举类型，此处使用 Number 类型代替
pack_params	Number	设备参数	由于 MVVM-JS 项目暂不支持自定义枚举类型，此处使用 Number 类型代替
io1	Boolean	设备数据	随机生成，由设备类型决定是否显示
io1	Boolean	设备数据	随机生成，由设备类型决定是否显示
a1	Number	设备数据	随机生成，由设备类型决定是否显示
a2	Number	设备数据	随机生成，由设备类型决定是否显示
temp	Number	设备数据	随机生成，由设备类型决定是否显示
tps	Number	设备数据	随机生成，由设备类型决定是否显示

注: 由于 JS 中暂不支持自定义枚举类型，本案例中”device”模型的 pack_type 属性和 pack_params 属性都采用 JS 中的 Number 类型代替，其中 pack_type 属性将决定该设备显示哪些数据，为了方便演示效果，这些设备数据均是随机生成的。

5.1.1 新增设备模型

根据案例需求，此处需要新增一个设备模型，名称为”device”，步骤详见上文 4.3.1 章节，效果如下图所示：

图 5.1 新增设备模型

在 AWTK Designer 新增 device 模型保存后，会自动在项目目录的”src/models”文件夹中生成 device 模型的相关代码，此处采用 JS 语言实现，模型代码非常简洁，其中 constructor() 函数为 device 的构造函数，在 new device 对象时会被调用：

```
/* src/models/device.js */
/**
 * @class device
 * @parent Object
 * @annotation ["model", "custom_prop"]
 */
export class device {
  constructor() {
    /**
     * @property {Number} pack_type
     * @annotation ["readable", "writable", "defvalue:0"]
     * 设备类型
     */
    this.pack_type = 0;

    /**
     * @property {Number} pack_params
     * @annotation ["readable", "writable", "defvalue:0"]
     * 设备参数
     */
    this.pack_params = 0;

    /**
     * @property {Boolean} io1
     * @annotation ["readable", "writable", "defvalue:false"]
     */
    this.io1 = false;

    /**
     * @property {Boolean} io2
     * @annotation ["readable", "writable", "defvalue:false"]
     */
    this.io2 = false;

    /**
     * @property {Number} a1
     * @annotation ["readable", "writable", "defvalue:0"]
     */
    this.a1 = 0;

    /**
     * @property {Number} a2
     * @annotation ["readable", "writable", "defvalue:0"]
     */
    this.a2 = 0;
```

```
    /**
     * @property {Number} temp
     * @annotation ["readable", "writable", "defvalue:0"]
     */
    this.temp = 0;

    /**
     * @property {Number} tps
     * @annotation ["readable", "writable", "defvalue:0"]
     */
    this.tps = 0;
  }
}
```

5.1.2 随机生成设备模型中的数据

在本案例中，为了方便演示，设备模型中的数据都是随机生成的，因此，我们可以修改 device 模型的构造函数，在 new device 对象时随机生成相应的数据，代码如下：

```
/* src/models/device.js */
export class device {
  constructor() {  /* device 模型的构造函数 */
    /* 属性注释详见上一小节，此处重点展示随机生成数据的代码 */
    this.pack_type = Math.round(Math.random() * 1000) % 15;
    this.pack_params = Math.round(Math.random() * 1000) % 13;
    this.io1 = Math.random() > 0.5;
    this.io2 = Math.random() > 0.5;
    this.a1 = Math.random() * 1000;
    this.a2 = Math.random() * 1000;
    this.temp = Math.random() * 1000;
    this.tps = Math.round(Math.random() * 1000);
  }
}
```

5.2 视图模型设计

完成模型设计后，需要为主界面 home_page 新建一个视图模型 home_page_view_model，步骤详见上文 4.2.1 章节。在 AWTK Designer 中新建并保存视图模型后，会自动在项目目录的”src/view_models”文件夹中生成 home_page_view_model.js 文件。

由于 JS 是脚本语言，可通过名字直接访问对象的成员变量和成员函数，因此视图模型的代码相较于 C 语言简洁很多，MVVM-JS 项目的 ViewModel 框架代码详见 4.2.1 章节。

需要注意的是，在 MVVM-JS 中，View 与 ViewModel 之间通过 ViewModel 的名称来绑定，例如此处的 ViewModel 名称为 home_page_view_model，在界面 UI 文件中可以通过 v-model 属性来指定，代码如下：

```
<!-- design/default/ui/home_page.xml -->
<window v-model="home_page_view_model" name="home_page" v-on:window_open="{reset}">
  ....
</window>
```

在 ViewModel 的 JS 代码文件中则可以通过 ViewModel() 函数来注册，代码如下：

```
/* src/view_models/home_page_view_model.js */
 /**
 * @class home_page_view_model
 * @parent ViewModel
 * @annotation ["model", "view_model", "custom_prop"]
 */
ViewModel('home_page_view_model', {/* home_page_view_model
↪对象，包含成员属性、成员函数（命令） */});
```

5.2.1 为视图模型增加属性

此处为了实现案例中的相关功能，需要为 home_page_view_model 添加以下属性：

属性名称	类型	默认值	描述
items	Array	[]	设备列表，用于存储 device 对象
unlocked	Boolean	false	用于解锁
currentIndex	Number	-1	当前选中设备序号，用于实现插入和删除功能

在 AWTK Designer 中添加完成后效果如下图所示：

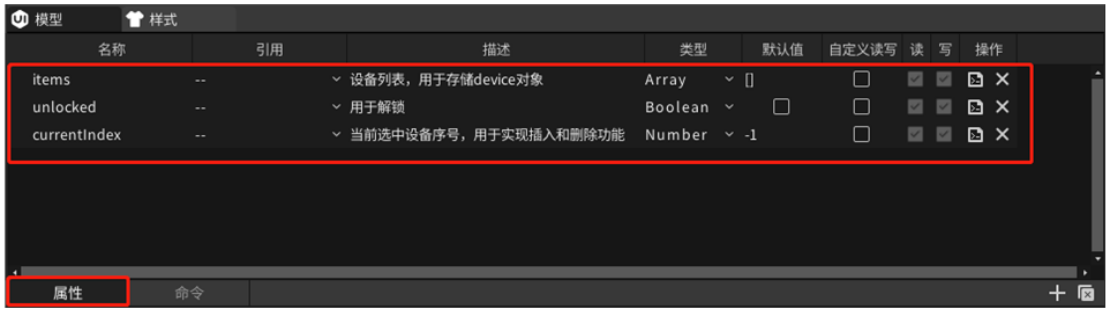

图 5.2 home_page_view_model 中的属性

保存视图模型后，在 home_page_view_model.js 文件中也会将以上属性自动添加到 ViewModel 对象的 data 中，代码如下：

```
/* src/view_models/home_page_view_model.js */
ViewModel('home_page_view_model', {
  data: {  /* ViewModel中的成员属性 */
    /**
     * @property {Array} items
     * @annotation ["readable", "writable", "defvalue:[]"]
     * 设备列表，用于存储device对象
     */
    items: [],
```

```
    /**
     * @property {Boolean} unlocked
     * @annotation ["readable", "writable", "defvalue:false"]
     * 用于解锁
     */
    unlocked: false,

    /**
     * @property {Number} currentIndex
     * @annotation ["readable", "writable", "defvalue:-1"]
     * 当前选中设备序号，用于实现插入和删除功能
     */
    currentIndex: -1
  },
  ......
});
```

5.2.2 为视图模型增加命令

此处为了实现案例中的相关功能，需要为 home_page_view_model 添加以下命令：

命令名称	描述
setCurrent	设置当前设备序号
insert	插入设备
remove	移除设备
clear	清空设备列表
reset	重置设备列表

在 AWTK Designer 中添加完成后效果如下图所示：

图 5.3 home_page_view_model 中的命令

保存视图模型后，在 home_page_view_model.js 文件中也会将以上属性自动添加到 ViewModel 对象的 methods 中，代码如下：

```
/* src/view_models/home_page_view_model.js */
ViewModel('home_page_view_model', {
  methods: {
    /**
     * @method remove
     * @annotation ["command"]
```

```
   * 移除设备
   * @param {String|Object} args 命令的参数。
   * @return {TRET} 返回RET_OK表示成功，否则表示失败。
   */
  canRemove: function (args) {
    return true;
  },
  remove: function (args) {
    return RET_OK;
  },

  /**
   * @method insert
   * @annotation ["command"]
   * 插入设备
   * @param {String|Object} args 命令的参数。
   * @return {TRET} 返回RET_OK表示成功，否则表示失败。
   */
  canInsert: function (args) {
    return true;
  },
  insert: function (args) {
    return RET_OK;
  },

  /**
   * @method clear
   * @annotation ["command"]
   * 清空设备列表
   * @param {String|Object} args 命令的参数。
   * @return {TRET} 返回RET_OK表示成功，否则表示失败。
   */
  canClear: function (args) {
    return true;
  },
  clear: function (args) {
    return RET_OK;
  },

  /**
   * @method reset
   * @annotation ["command"]
   * 重置设备列表
   * @param {String|Object} args 命令的参数。
   * @return {TRET} 返回RET_OK表示成功，否则表示失败。
   */
  canReset: function (args) {
    return true;
  },
```

```
    reset: function (args) {
      return RET_OK;
    },

    /**
     * @method setCurrent
     * @annotation ["command"]
     * 设置当前设备序号
     * @param {String|Object} args 命令的参数。
     * @return {TRET} 返回RET_OK表示成功，否则表示失败。
     */
    canSetCurrent: function (args) {
      return true;
    },
    setCurrent: function (args) {
      return RET_OK;
    }
  },
  ......
});
```

接下来，我们逐个实现以上命令的功能。

5.2.3 实现设置当前设备序号功能

首先是 setCurrent 命令，该命令始终可用（没有条件限制），用于设置设备列表的当前选中项，即点击列表中某个设备，将它的索引保存到 ViewModel 的 currentIndex 属性中，代码如下：

```
/* src/view_models/home_page_view_model.js */
ViewModel('home_page_view_model', {
  methods: {
    canSetCurrent: function (args) {
      return true;  /* setCurrent 命令始终可用，因此直接返回 true */
    },
    setCurrent: function (args) {     /* 执行命令时传入参数：设备索引（index） */
      console.log(args.index);        /* 打印设备索引 */
      this.currentIndex = args.index; /* 将设备索引保存到 currentIndex 属性 */
      this.notifyPropsChanged()       /* 通知界面更新 */
      return RET_OK;
    }
  },
  ......
});
```

5.2.4 实现插入设备功能

insert 命令用于在当前设备列表的选中项前插入新项，该命令同样是只有当选中项索引在设备列表数组中的时候才可用，代码如下：

注: 这里插入的数据是 device 对象，因此需要先导入上文 5.1.1 章节中新增的 device 模型。

```
/* src/view_models/home_page_view_model.js */
import { device } from "../models/device";  /* 导入 device 模型 */

ViewModel('home_page_view_model', {
  methods: {
    canInsert: function (args) {
      /* 获取当前选中项索引 */
      var index = this.currentIndex;
      /* remove 命令只有当选中项索引在设备列表数组中的时候才可用 */
      return index >= 0 && index <= this.items.length;
    },
    insert: function (args) {
      var index = this.currentIndex;       /* 获取当前选中项索引 */
      var item = new device;               /* 创建 device 对象 */
      this.items.splice(index, 0, item);   /* 插入数据 */
      this.notifyItemsChanged(this.items); /* 通知界面更新 */
      return RET_OK;
    },
  },
  ......
});
```

5.2.5 实现移除设备功能

remove 命令用于移除当前设备列表中的选中项，该命令只有当选中项索引在设备列表数组中的时候才可用，代码如下：

```
/* src/view_models/home_page_view_model.js */
ViewModel('home_page_view_model', {
  methods: {
    canRemove: function (args) {
      /* 获取当前选中项索引 */
      var index = this.currentIndex;
      /* remove 命令只有当选中项索引在设备列表数组中的时候才可用 */
      return index >= 0 && index < this.items.length;
    },
    remove: function (args) {
      var index = this.currentIndex;       /* 获取当前选中项索引 */
      this.items.splice(index, 1);         /* 删除设备列表中的选中项 */
      this.notifyItemsChanged(this.items); /* 通知界面更新 */
      return RET_OK;
    },
  ......
```

```
  }
});
```

5.2.6 实现清除设备列表功能

clear 命令用于清除设备列表，即删除设备列表中的所有数据，它只有在设备列表中存在数据时可用，代码如下：

```
/* src/view_models/home_page_view_model.js */
ViewModel('home_page_view_model', {
  methods: {
    canClear: function (args) {
      /* clear 命令只有在设备列表中存在数据时可用 */
      return this.items.length > 0;
    },
    clear: function (args) {
      this.items.splice(0, this.items.length); /* 清除设备列表中的所有数据 */
      this.notifyItemsChanged(this.items);     /* 通知界面更新 */
      return RET_OK;
    },
  },
  ......
});
```

5.2.7 实现重置设备列表功能

reset 命令用于重置设备列表，此处是向设备列表中添加 50 条数据，该命令只有在设备列表中没有数据时才可用，代码如下：

注: 这里插入的数据是 device 对象，因此需要先导入上文 5.1.1 章节中新增的 device 模型。

```
/* src/view_models/home_page_view_model.js */
import { device } from "../models/device";  /* 导入 device 模型 */

ViewModel('home_page_view_model', {
  methods: {
    canReset: function (args) {
      /* reset 命令只有在设备列表中没有数据才可用 */
      return this.items.length == 0;
    },
    reset: function (args) {
      /* 向设备列表插入50条数据 */
      for (var i = 0; i < 50; i++) {
        var item = new device;                          /* 创建 device 对象 */
        this.items.splice(this.items.length, 0, item); /* 插入数据 */
      }
      /* 通知界面更新 */
      this.notifyItemsChanged(this.items);
      return RET_OK;
    },
```

```
  },
  ......
});
```

5.3 界面设计

完成 Mvvm-JS-Demo 案例的 Model 和 ViewModel 后，接下来需要进行界面设计。**由于 Mvvm-JS-Demo 的界面与 Mvvm-C-Demo 完全一致，因此读者只需参考上文 3.3 章节的内容即可**，此处就不过多赘述了。

注: 这里也体现了 MVVM 项目的核心优点：界面与业务逻辑之间的松耦合，它们均可独立变化，二者通过一条条绑定规则进行串联。

5.4 运行效果

MVVM-JS 项目的运行过程跟 MVVM-C 项目略有不同，具体详见下文。

5.4.1 打包资源

在 AWTK Deisgner 中，打包 MVVM-JS 项目资源的步骤与 MVVM-C 项目一样，点击菜单栏中的”打包”按钮即可，如下图所示：

图 5.4 打包资源

但 AWTK Designer 在打包 MVVM-JS 项目的资源时，会将 src 下的所有 JS 脚本打包成一个文件，放在资源目录的 scripts 文件夹中，名称为 app.js。

例如此处会将 Mvvm-JS-Demo/src 文件夹中的 JS 文件打包为 res/assets/default/raw/scripts/app.js。在项目运行时，将直接读取这个 app.js 文件来执行里面的函数。

5.4.2 模拟运行

A0WTK Designer 内置了 MVVM-JS 项目的模拟程序，因此，MVVM-JS 项目无需编译即可查看效果，如下图所示，点击菜单栏中的”模拟运行”：

图 5.5 模拟运行

运行效果如下图所示：

图 5.6　Mvvm-JS-Demo

5.4.3　编译运行

MVVM-JS 项目的业务逻辑由 JS 脚本实现，程序的 main() 函数只需完成 AWTK、AWTK-MVVM 和 JerryScript 的初始化工作即可，因此实际上 MVVM-JS 项目的启动程序代码是一样的，详见项目的 scr/main.c 和 src/application.c 文件，它们由 AWTK Designer 自动生成。

由于以上特性，AWTK Designer 中不支持直接编译 MVVM-JS 项目，而是内置了一个模拟程序，即 MVVM-JS 项目的启动程序（runFlowAWTK），用来模拟运行 MVVM-JS 项目，详见上一小节。

用户如果想自行编译 MVVM-JS 项目，在项目目录下打开终端，执行 scons 命令即可，如下图所示：

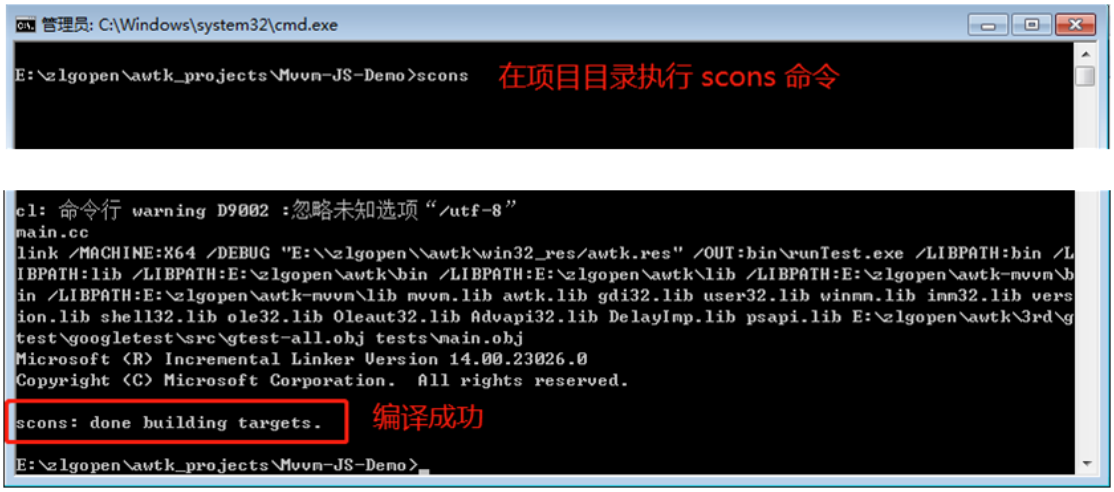

图 5.7　编译 Mvvm-JS-Demo

编译成功后，可在项目中看到的 bin/demo.exe 文件（此处以 Windows 平台为例），双击

运行的效果与模拟运行一致，详见上一小节中的效果图。

注：如果 MVVM-JS 项目中使用了 AWFlow Designer 设计的流图（即开发 AWTK+AWFlow 低代码应用程序），则必须使用 AWTK Designer 中的模拟运行（runFlowAWTK）查看效果，使用 scons 命令自行编译的程序默认关闭流图功能，关于 AWTK+AWFlow 低代码应用程序的相关介绍请参考：《AWFlow+AWTK 低代码应用开发指南》。

6. 高级功能

6.1 MVVM 控制动画启停

MVVM 模式中，为了彻底分离用户界面与业务逻辑，通常不能在业务代码中调用 GUI 相关的 API 控制 UI 动画的启停。为此，AWTK 提供了一个特殊的解决方案，其原理是通过指定控件（widget_t）中的“exec”属性来执行特定的函数，“exec”属性的值由函数名和参数两部分组成，两者用英文冒号分隔，目前主要用于动画的控制，支持的函数有：

函数名	说明
start_animator	开始动画
stop_animator	停止动画
pause_animator	暂停动画
destroy_animator	销毁动画

比如下面的属性表示开始 rotation 动画，其中 rotation 为动画名称，如果不指定动画名称，则表示开始该控件上的所有动画：

```
exec="start_animator:rotation"
```

在 MVVM 项目中，一般是根据模型中数据来控制动画。比如模型中有一个 value 数据，当 value 大于 3 时，启动 label 控件上的所有动画，否则暂停控件上的所有动画，UI 文件代码如下：

```
<label x="0" y="0" w="160" h="28" text="Label" v-data:exec="{value > 3 ? 'start_
↪animator' : 'pause_animator'}" animation="opacity(easing=linear,auto_start=false,
↪from=0,to=255)"/>
```

在 AWTK Designer 中只需对控件的“exec”属性进行数据绑定即可，操作如下图所示：

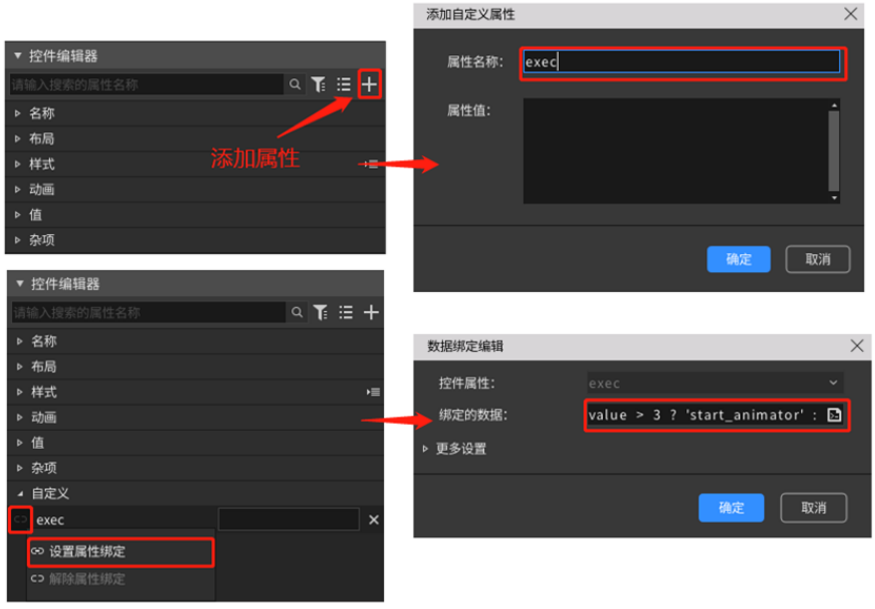

图 6.1　添加 exec 属性并进行数据绑定

注：上述采用了数据绑定的方式更新“exec”属性的值，当模型中的 value 变化时，会触发 ViewModel 更新，此时会执行“exec”中的代码。

此外，在 AWTK Designer 中，我们还可以通过调用 FScript 脚本中的 widget_set() 函数来主动设置控件的“exec”属性，步骤如下：

步骤一：创建一个 button 控件，并为 button 控件创建一个控件动画，动画名称为”move”，如下图所示：

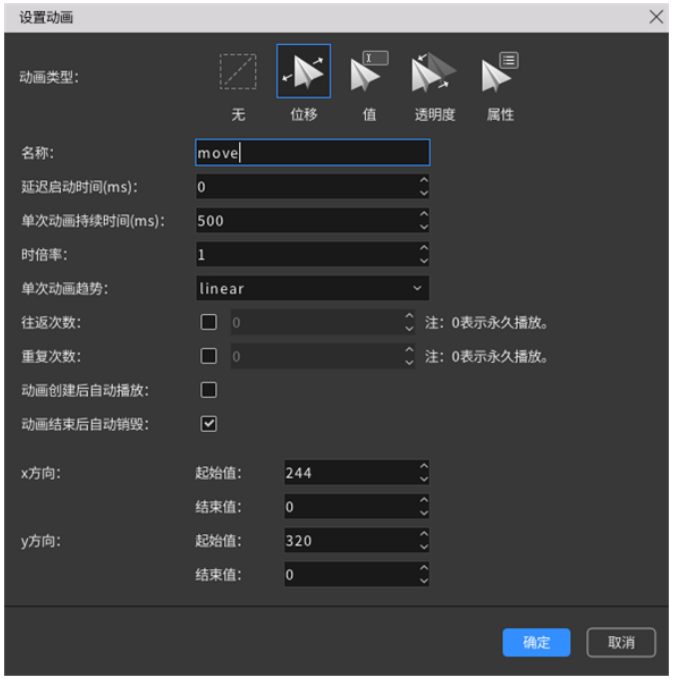

图 6.2　创建控件动画

步骤二：为 button 控件添加一个点击事件，并在点击事件中执行 FScript 脚本，如下图所示：

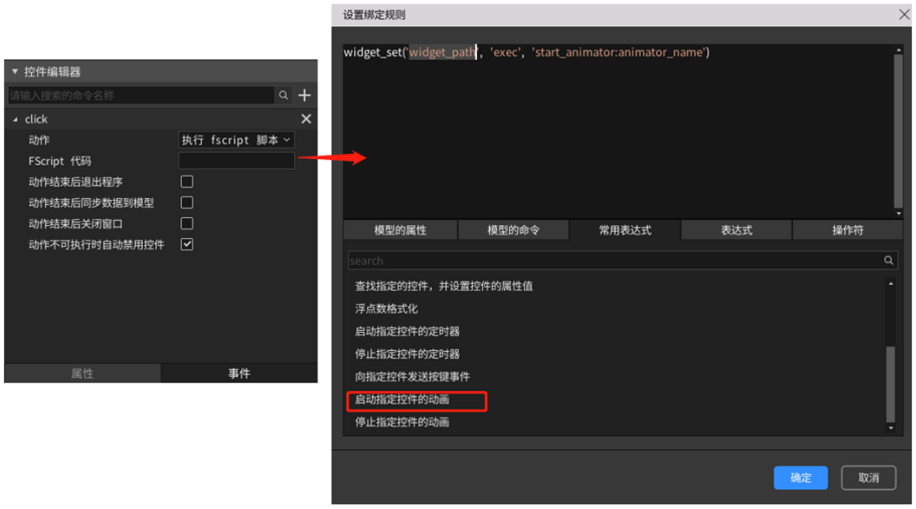

图 6.3　添加事件

步骤三：编写 FScript 脚本启动控件上的”move”动画，代码如下：

```
/* 设置当前(self)控件上的"exec"属性，属性值为"start_animator:move" */
widget_set('self', 'exec', 'start_animator:move')
```

注: widget_set() 接口的声明及其用法请参考 FScript 的相关文档。

步骤四：点击 button 控件，可以看见”move”动画开始播放。

上述示例只介绍了启动动画（start_animator），其他动画函数的用法也是类似的。

诚信共赢，持续学习，客户为先，专业专注，只做第一

广州致远电子股份有限公司

更多详情请访问 www.zlg.cn

欢迎拨打全国服务热线 400-888-4005